AD-A041 237    GENERAL RESEARCH CORP SANTA BARBARA CALIF    F/G 9/2
JAVS FINAL REPORT.(U)
JUN 77  C GANNON, N B BROOKS, W R WISEHART    F30602-76-C-0233
UNCLASSIFIED    RADC-TR-77-201    NL

1 OF 1
AD
A041237

END
DATE
FILMED
7-77

12

RADC-TR-77-201
Final Technical Report
June 1977

# JAVS FINAL REPORT

General Research Corporation

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

APPROVED: Frank S LaMonica
FRANK S. LAMONICA
Project Engineer

APPROVED: [illegible]
ROBERT D. KRUTZ, Colonel, USAF
Chief, Information Sciences Division

FOR THE COMMANDER: John P. Huss
JOHN P. HUSS
Acting Chief, Plans Office

| | |
|---|---|
| 19 REPORT DOCUMENTATION PAGE | READ INSTRUCTIONS BEFORE COMPLETING FORM |
| 18 1. REPORT NUMBER: RADC-TR-77-201 / 2. GOVT ACCESSION NO. | 3. RECIPIENT'S CATALOG NUMBER: 9 |
| 4. TITLE (and Subtitle): 6 JAVS FINAL REPORT | 5. TYPE OF REPORT & PERIOD COVERED: Final Technical Report, May 76 - Jan 77 |
| | 6. PERFORMING ORG. REPORT NUMBER: N/A |
| 7. AUTHOR(s): 10 C. Gannon, N. B. Brooks, W. R. Wisehart | 8. CONTRACT OR GRANT NUMBER(s): 15 F30602-76-C-0233 |
| 9. PERFORMING ORGANIZATION NAME AND ADDRESS: General Research Corporation, P.O. Box 3587, Santa Barbara CA 93105 | 10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 16 63728F 17 55500838 18 |
| 11. CONTROLLING OFFICE NAME AND ADDRESS: Rome Air Development Center (ISIM), Griffiss AFB NY 13441 | 12. REPORT DATE: 11 June 1977 |
| | 13. NUMBER OF PAGES: 71 |
| 14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Same 12 76p. | 15. SECURITY CLASS. (of this report): UNCLASSIFIED |
| | 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE: N/A |

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Frank S. LaMonica (ISIM)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Computer Software
Software Testing
Software Verification
JAVS
Automated Verification System

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

As part of its program to improve software quality and reliability through the implementation of advanced technology, Rome Air Development Center has contracted with General Research Corporation for assistance in using GRC's JOVIAL Automated Verification System (JAVS) to structurally test a large and complex operational program not designed for automated testing. While performing the systematic test, JAVS was evaluated and enhanced, and an applicable testing methodology was developed.

(Cont'd)

402 754

This report describes the results of the systematic test, evaluation of JAVS, testing methodology current configuration of JAVS, and proposed capabilities of future tools.

JAVS is operational on the HIS 6180 at RADC operating under GCOS, the HIS 6080 at Offutt AFB, Omaha, Nebraska under WWMCCS, and the CDC 6400 at GRC in Santa Barbara, California under the GOLETA Operating System.

## LIST OF JAVS REPORTS

- JAVS Technical Report: Vol. 1, User's Guide. This report is an introduction to using JAVS in the testing process. Its primary purpose is to acquaint the user with the innate potential of JAVS to aid in the program testing process so that an efficient approach to program verification can be undertaken. Only the basic principles by which JAVS provides this assistance are discussed. These give the user a level of understanding necessary to see the utility of the system. The material on JAVS processing in the report is presented in the order normally followed by the beginning JAVS user. Adequate testing can be achieved using JAVS macro commands and the job streams presented in this guide. The Appendices include a summary of all JAVS commands and a description of JAVS operation at RADC with both sample command sets and sample job control statements. (RADC-TR-77-126, Vol I dated April 77).

- JAVS Technical Report: Vol. 2, Reference Manual. This report describes in detail JAVS processing and each of the JAVS commands. The Reference Manual is intended to be used along with the User's Guide which contains the machine-dependent information such as job control cards and file allocation. Throughout the Reference Manual, modules from a sample JOVIAL program are used in the examples. Each JAVS command is explained in detail, and a sample of each report produced by JAVS is included with the appropriate command. The report is organized into two major parts: one describing the JAVS system and the other containing the description of each JAVS command in alphabetical order. The Appendices include a complete listing of all error messages directly produced by JAVS processing. (RADC-TR-77-126, Vol II dated April 77).

- JAVS Technical Report: Vol. 3, Methodology Report. This report describes the methodology which underlies and is supported by JAVS. The methodology is tailored to be largely independent of implementation and language. The discussion in the text is intended to be intuitive and demonstrative. Some of the methodology is based upon the experience of using JAVS to test a large information management system. A long-term growth path for automated verification systems that supports the methodology is described. (RADC-TR-77-126, Vol III dated April 77).

- JAVS Computer Program Documentation: Vol. 1, System Design and Implementation. This report contains a description of JAVS software design, the organization and contents of the JAVS data base, and a description of the software for each JAVS component: its function, each of the modules in the component, and the global data structures used by the component. The report is intended primarily as an informal reference for use in JAVS software maintenance as a companion to the Software Analysis reports described below. Included in the appendices are the templates for probe code inserted by instrumentation processing for both structural and directive instrumentation and an alphabetical list of all modules in the system (including system routines) with the formal parameters and data type of each parameter.

- JAVS Computer Program Documentation: Vol. 2, Software Analysis. This volume is a collection of computer output produced by JAVS standard processing steps. The source for each component of the JAVS software has been analyzed to produce enhanced source listings of JAVS with indentation and control structure identification, inter-module dependence, all module invocations with formal and actual parameters, module control structure, a cross reference of symbol usage, tree report for each leading module, and report showing size of each component. It is intended to be used with the System Design and Implementation Manual for JAVS software maintenance. The Software Analysis reports, on file at RADC, are an excellent example of the use of JAVS for computer software documentation.

- JAVS Preprocessor for JOVIAL. This report, prepared for GRC by its subcontractor, System Development Corporation (SDC), describes the software for the JAVS-2 component: its origin as the GEN1 part of the SAM-D ED Compiler, the modifications made in GEN1 to adapt the code for JAVS-2, the JAVS-2 code modules, and the data structures. It contains excerpts of other SDC reports on the SAM-D ED JOVIAL Compiler System. The report reflects the status of the software for JAVS-2 as delivered by SDC to GRC in September 1974. The description of JAVS-2 software contained in the System Design and Integration report reflects the status of JAVS-2 as delivered to RADC by GRC in September 1975 and thereby supercedes the SDC report.

- JAVS Final Report. The final report for the project describes the implementation and application of a methodology for systematically and comprehensively testing computing software. The methodology utilizes the structure of the software undergoing test as the basis for anlaysis by an automated verification system (AVS). The report also evaluates JAVS as a tool for software development and testing.

CONTENTS

## EVALUATION

The purpose of this effort was to enhance the JOVIAL Automated Verification System (JAVS) and to implement a systematic software testing program using the JAVS to assist in the testing process. Developed to aid in the testing and verification of JOVIAL J3 programs, it provides the ability to increase the practical reliability of software by increasing the achieved level of testing. As a result of this effort, the JAVS was enhanced and successfully tuned for operational use.

Excellent supporting documentation on its application and use is available.

Frank S LaMonica
FRANK S. LAMONICA
Project Engineer

# 1. INTRODUCTION

JAVS is a software tool intended to be applied during testing of JOVIAL J3 programs to aid in the recognition of untested program paths, to assist in the development of additional test cases appropriate to improvement of testing coverage, and to automatically document the computer program. Program verification is provided by analysis of program control structures, instrumentation of the program through insertion of software probes to measure testing coverage during execution, and comprehensive reports which pinpoint paths in the program structure that remain to be exercised. Retesting guidance is provided by the identification of program paths that lead to the untested areas of program and by reports describing module and symbol interaction.

In addition to program testing, JAVS can be used to assist with program debugging, system integration, and software maintenance. JAVS provides computation directives to aid in such aspects of debugging as checking array sizes, the values of logical expressions, and boundaries of variables. The directives are a special JOVIAL comment expanded by JAVS into executable source code during instrumentation. System integration and software maintenance are supported by JAVS's analysis of module interaction. The system-level analysis, as well as the control structure analysis performed at the individual module level, provides a basis for a variety of documentation reports. Reports can be obtained which show module control structure, module invocation interaction and hierarchy, unused modules, and a system-wide symbol cross reference, to name a few of the automated reports. Self-documentation was performed on the JAVS software and has been found to be exceedingly useful in maintaining and expanding JAVS.

JAVS was developed under contract with Rome Air Development Center.[*] The specific tasks of that contract were to engineer workable and practical first-level solutions to automating the measurement of computer program testing effectiveness, assistance in manual test case design and selection, and increased mechanization of certain aspects of software system maintenance.

Under the current JAVS Implementation contract with RADC,[**] JAVS was to be utilized in a systematic test in order to evaluate its performance as a testing tool.

## 1.1 OBJECTIVES

There were three major objectives in the current one-man-year level-of-effort contract:

1. Stress JAVS as much as possible to evaluate its performance by implementing a systematic software test on a large and complex operational program not written with Automated Verification System (AVS) testing in mind.

[*] Air Force Contract F30602-73-C-0344

[**] Air Force Contract F30602-76-C-0233

2. Correct any flaws found in JAVS, and implement whatever enhancements are feasible within the contract level-of-effort.

3. Improve the existing testing methodology[1] to reflect the experience gained during the current testing process.

Evaluation of JAVS by testing an operational program with it was of particular importance because it is the first operational use of JAVS, a program whose initial acceptance was based on structure-based tests (JAVS-self-test), with definite goals for statement coverage in the tests. In addition, the tests demonstrate the utility of JAVS in particular and Automated Verification Systems in general to support software testing. It provided exposure to JAVS's strengths and weaknesses and was the transition period into the "real world" for JAVS. Finally, the experience gained from JAVS processing large amounts of source code written without provision for AVS testing would provide the basis for establishing a comprehensive and procedural methodology for software testing and determine requirements for AVS-supported testing.

## 1.2 ENGINEERING SERVICES

As part of the contract, and in addition to the testing effort, GRC performed the following specific engineering tasks:

- Combined the six separate JAVS programs into a single overlay program
- Conducted a two-week training course
- Provided on-site assistance to the Air Force as required
- Explored and implemented modifications to JAVS to make it more applicable to the tester's needs
- Reviewed the testing process with Air Force personnel periodically
- Maintained a log of day-to-day operations

Services provided in addition to those specified included the development and installation of a macro command processor supported by the JAVS overlay program and assistance in installing JAVS at SAC Headquarters, Offutt AFB, on the HIS 6080 operating under the WWMCCS operating system.

## 1.3 JAVS TECHNICAL REPORTS

This report describes the objectives and activities undertaken by GRC in the course of its JAVS Implementation contract with RADC. Section 2 contains a description of the systematic software test which provides the basis for an evaluation of JAVS capabilities, described in Sec. 3; a software testing methodology, described in Sec. 4; and a description of JAVS processing, organization, operation, and enhancements made during this contract in Sec. 5. A forecast of future Automated Verification Systems is contained in Sec. 6 which includes static and dynamic tools and the requirements of new AVS.

The JAVS Methodology Report[2] contains an in-depth, detailed methodology for comprehensive testing of software using an AVS. This report describes a methodology which is primarily independent of a specific AVS tool, although an overview of JAVS and the testing experience of the current contract is provided.

Two documents describing the capabilities and operation of JAVS were written: the JAVS User's Guide[3] and JAVS Reference Manual[4]. These manuals contain the information needed to use the current configuration of JAVS and supercede the two manuals of the same names written for the JAVS Development contract.

The computer program documentation consists of two reports: (1) a description of JAVS's software design, data base, and the function of each module and component[5], and (2) a volume containing computer output (JAVS self-documentation) describing JAVS's control structure, module interaction, symbol usage, and summary data on source code size and component definition.

## 2 OPERATIONAL TESTS WITH JAVS

This section describes the tests conducted on a large component of the Force Management Information System (FMIS), which is a generalized data management system in regular use by the Air Force. One component, called COMPOSE, was selected as sufficiently large and complex to adequately exercise JAVS. Tests were conducted on COMPOSE using JAVS to aid in understanding the program organization, and as the means of measuring test effectiveness in terms of the percent of the program's logical segments exercised by the tests.

### 2.1 DESCRIPTION OF COMPOSE

The FMIS computer program was selected by the Air Force for testing primarily because of its size and complexity. FMIS is a highly modular software package which can operate in a variety of modes: real-time on-line, time-share, interactive batch, and off-line batch. The primary functions that FMIS performs are to create, maintain, update and make retrievals into a data base. The retrieval component consists of two subcomponents: COMPOSE and QUERY. COMPOSE, which was selected for thorough testing, recognizes some of the FMIS commands and performs data retrieval, data base field calculations, and report formatting. The input data for FMIS are a data base and a set of commands which affect the data base and generate reports.

FMIS was installed at RADC in object form for the purpose of this test. The source code for the executable modules of COMPOSE was provided for input to the JAVS syntax analyzer, but the text of the COMPOOLs used in compiling the source was not provided. A model FMIS data base and a set of commands constituting an initial test case were also supplied.

### 2.2 TEST PROCEDURES

A briefing on COMPOSE was given to the test team at the beginning of the test period. This briefing described the organization of COMPOSE at a high level. The test team was to use JAVS to document the organization of COMPOSE, and to perform the instrumentation and test analysis.

The first step was to develop a test plan with emphasis on insuring that the tests met the goals of stressing JAVS as well as testing COMPOSE. The COMPOSE code was then entered into a set of JAVS libraries, documentation reports were generated, the code was instrumented, tests were run, and the results analyzed.

#### 2.2.1 Development of a Test Plan

The test plan was developed early in the contract as part of a two-week JAVS training course conducted by GRC at RADC. At that time the primary interest was to determine any flaws in JAVS by thoroughly stressing it through exposure to large amounts of JOVIAL code. The test plan is included as Appendix A.

The main steps in testing COMPOSE were:

- Process all of COMPOSE in one library to determine the complete invocation hierarchy
- Use the FMIS data base and input commands supplied by SAC for the initial test
- Analyze structural execution coverage reports produced by JAVS, and retest COMPOSE as necessary

### 2.2.2 Building the Library

The first attempt at testing COMPOSE started with building a single JAVS library containing all of the COMPOSE source code. This was not possible because the number of invokable modules exceeded the maximum permitted by JAVS on a single library (250 modules). The test plan was altered to the extent that COMPOSE was partitioned into seven components with only minor overlap, using an abbreviated description of the software functions provided by the Air Force as a guide. The components ranged in size from 52 to 218 modules; the seven component libraries that resulted comprised a total of 705 invokable modules and 38,734 JOVIAL statements, excluding comments.

### 2.2.3 JAVS Processing of COMPOSE

Each of the COMPOSE components was processed by the JAVS syntax and structural analyzers. At this point the JAVS documentation reports for the libraries were obtained to facilitate understanding the source code. The reports processed were the enhanced module listing (PRINT, MODULE*), DD-path (decision-to-decision path) definition listing (PRINT, DDPATHS*), control flow picture (ASSIST,PICTURE*), library module interaction matrix (DEPENDENCE, GROUP,LIBRARY*), auxiliary library module interaction matrix (DEPENDENCE, GROUP,AUXLIB*), symbol cross reference (ASSIST,CROSSREF,LIBRARY*), eleven-level intermodule calling tree (DEPENDENCE,BANDS*), library summary report (DEPENDENCE, SUMMARY*), and library statistics report (provided at the end of each JAVS run).

Documentation reports were obtained for all seven COMPOSE components, and copies were sent to SAC. In the time remaining for testing, three of the components were instrumented, executed, and analyzed. At SAC Headquarters, following installation of JAVS operating under WWMCCS on the HIS 6080, a fourth component containing 72 modules and appropximately 3,200 JOVIAL statements was processed by JAVS. This effort produced documentation reports, instrumentation of the source code, execution of FMIS (including the instrumented code) with supplied data, and post-test analysis and trace reports.

*The output for each of these JAVS reports is shown in the JAVS Reference Manual under each of these JAVS commands.

### 2.2.4 Executing the Tests

FMIS normally operates in an interactive environment under the time-sharing facilities of the operating system. The user supplies a data base, enters a series of commands to FMIS, and receives responses from FMIS on an interactive terminal. This type of interaction is not always possible, nor is it necessarily desirable, for test purposes. The overhead of code expansion through instrumentation as well as the additional processing time can be prohibitive under the time-sharing constraints. Furthermore, the need to record and correlate all input data with test results, and possibly to duplicate a test precisely, demands a controlled test environment. For these reasons, COMPOSE was tested in batch mode.

However, batch mode operation presented additional problems of its own. Because the software was self-protected against erroneous data (i.e., error routines were invoked and execution terminated), it was difficult to prepare data that resulted in tests with normal software termination. The solution to this problem was to generate test data and first input it to the standard (interactive) version of COMPOSE. When normally terminating command streams were found, they were then used as inputs to the instrumented version running in the batch mode.

Executing FMIS with instrumented source code produces the normal FMIS output and a JAVS trace file containing information accumulated for each DD-path exercised and each module invoked during the test execution. This file was analyzed to measure test effectiveness in terms of how many DD-paths were exercised, and to identify untested DD-paths for retesting.

The JAVS trace file is used as input to the JAVS post-test analyzer. The information it contains is analyzed to produce coverage reports at the DD-path and statement levels; execution trace reports at the module invocation and DD-path levels; a summary report showing percentage of DD-paths exercised for all instrumented modules; and reports showing which DD-paths were hit and which were not (each DD-path has a unique identification number). These reports are generated for each test case executed in the test case set and cumulatively for the entire execution run.

### 2.2.5 Analyzing Test Results

Analysis of the post-test summary report for each of the three COMPOSE components tested at RADC showed that a substantial number of small modules (those with few DD-paths) were never invoked during execution of the initial test case data set supplied with the source code. Inspection of the module description manual for COMPOSE showed that a significant number of the unexercised modules handled error conditions. Analysis of several JAVS documentation reports, primarily the control-indented module listing, component symbol cross reference, and module invocation report, showed that invocation of several of the error-handling modules would result in termination of the test, with only a very small improvement to the total testing coverage. Since stressing JAVS was a major goal, many of these tests were identified but not conducted.

It should be noted that program termination did not take place within the code comprising COMPOSE. At various places in the COMPOSE source, branches were made to modules not included in COMPOSE. In some cases, control would return to COMPOSE; in other cases execution would terminate before returning to COMPOSE. This design feature of FMIS strongly supports the testing requirement that the source code for the entire program be available to the test team, since satisfactory test analysis requires instrumentation of test termination points.

The DD-path and statement coverage reports showed which modules had poor coverage from the initial test. The initial test for each of the three components which were instrumented and tested achieved overall DD-path coverages of 15%, 29%, and 48%, respectively. Each component contained 1,150-1,400 DD-paths. It was evident from the coverage reports that many DD-paths were null switch points.*

At this point in the test period an assessment was made concerning retesting the components to achieve better DD-path coverage than was produced by the original test data. This assessment was based on the following information:

- Execution trace reports showed that much of the input data underwent substantial processing before it entered the test object.

- The component-wide cross reference showed that many variables affecting the selection of unhit DD-paths were global. Without COMPOOLs and the source listing for the rest of FMIS, these variables could not be tracked back to the input.

- JAVS documentation reports for the components showed numerous errors, primarily in describing module interaction, in the module description manual for COMPOSE. Reliance on this manual for module functional description was questionable.

Typical testing strategies for COMPOSE presented the following problems:

- Top-down testing proceeds from the modules highest on the invocation to those lowest in the structure for the complete software system. Since execution control branched into and out of COMPOSE, the testers had no control over manipulations of input data before it reached the targets for subsequent testing.

- Bottom-up testing proceeds from the modules at the bottom of the invocation structure to those highest in the structure. This testing strategy would require special testing environments which would directly invoke each level of modules, from the lowest to the highest in the structure. This strategy would solve the problem of data manipulation associated with top-down testing, but

---

*JOVIAL index and item switches can be coded to contain null points per the JOCIT Compiler User's Manual.[6]

JAVS module listings and tracing reports showed that control transferred between modules not only due to module invocations but also due to transfers to labels not residing in the module. Thus the "special testing environment" must include invoked modules and modules entered via label transfers. The special testing environment modules can be dummy modules, but the effort expended in building testing environments for large components (52-218 modules) can be considerable.

In view of these problems and the limited testing resources, the strategy selected for retesting the components was to choose as testing targets (1) the uninvoked modules which were not expected to cause program termination, and (2) unexercised DD-paths in high-level modules (those at the top of the invocation structure of COMPOSE) which resided deep in the control level of the modules. JAVS DD-path and statement reports show control structures indented according to their control level. Executing a deep-nested DD-path will collaterally execute some of the DD-paths leading to that level.

In order to determine which DD-paths were executed by each FMIS input data command (see Sec. 2.1), test case boundaries (defined by an invocation to a JAVS data collection routine in the instrumented source code) were placed wherever a new input command was recognized by COMPOSE. With the JAVS tracing and DD-path HIT report, the affect of each input command could thus be observed in detail.

The initial test case data was modified on the basis of the JAVS-reported program behavior so that as many test targets (modules uninvoked and deeply nested unexercised DD-paths) as possible would be executed and overall DD-path coverage improved. Time permitted retesting two of the three components that had been tested with initial test case data. Overall coverage in one component increased from 29% to 41%; coverage in the other increased from 48% to 54%.

Analysis of post-test results from retesting showed that many of the remaining DD-paths were designed to handle error conditions, boundary conditions, improbable input command combinations, or a more complex FMIS data base than that used during JAVS-supported testing.

## 2.3 SUMMARY OF COMPOSE TEST

The following section discusses conclusions drawn from the testing of COMPOSE. The conclusions relating to JAVS evaluation and Testing Methodology are found in Secs. 3 and 4, respectively.

### 2.3.1 Testing Conditions

Testing COMPOSE provided one measure of the effort required to structurally test a large, complex JOVIAL program under the following conditions:

1. The test team was unfamiliar with the test object's full function and operation, and available documentation did not fill in all the missing details.

2. The test object was available in source form, but not the source code of the global data (COMPOOLs) or the remainder of the larger program, which was intricately coupled to the test object.

3. JAVS itself was found to have errors not previously discovered. These had to be corrected before building JAVS libraries for the COMPOSE components needed for subsequent JAVS processing.

### 2.3.2 Test Conclusions

Three COMPOSE components were structurally tested at RADC (187 modules, 11,015 JOVIAL statements, excluding comments, and 4,518 DD-paths). One COMPOSE component was tested at SAC Headquarters (72 modules, 3,200 JOVIAL statements). DD-path coverage achieved for the four components was 15%, 41%, 54%, and 52%. The middle two percentages reflect retesting with data generated with JAVS assistance on the basis of the experience gained during testing. Analysis of the test results formed the following conclusions:

1. There was no structurally dead code in COMPOSE (i.e., code that is unreachable because the statements or modules are completely disconnected from the rest of the program).

2. Incomplete knowledge of the total program, and lack of testing resources (including the rest of the source program) to accomplish 100% path coverage did not permit identification of logically dead code (i.e., code that is unreachable because of some logical inconsistency in the values required to reach a path.)

3. For the coverage measured there were no catastrophic failures of COMPOSE; however, evaluation criteria were not available to check complete functional accuracy of the results.

4. The COMPOSE code was not sufficiently modular to support modular testing (top-down or bottom-up), nor did it permit full-scale testing of a small portion.

Although the tests were only the bare beginning of a thorough test of COMPOSE, the exercise provided substantial insight into JAVS utility and test methodology. These are discussed in the next two sections.

## 3 JAVS EVALUATION

Under the JAVS development contract (F30602-73-C-0344) completed in January 1976, JAVS was subjected to a systematic testing program that included not only functional demonstration but also a requirement that acceptance tests should exercise at least 85% of the executable source statements. The results of tests conducted at that time were reported in the JAVS Acceptance Tests for RADC, a GRC Internal Memorandum[7] and all the mistakes that were found during the self-test were corrected before final delivery of the JAVS System. The JAVS self-test is reviewed briefly in Sec. 3.1, following.

The major objective of the present contract was the further evaluation of JAVS under operational conditions, i.e., while conducting tests of an operational program. During these tests the study team would (1) verify that all JAVS processing functions performed correctly, (2) correct any mistakes found during evaluation, (3) attempt to understand how these mistakes went undetected during self-test, and (4) review the problem uncovered to see if they suggest changes in JAVS or only changes in the way it is used.

Viewed as a stressing exercise for JAVS, this operational test is more than an additional series of tests because it provides a basis for evaluating the effectiveness of the original structure-based testing (the JAVS self-test) in establishing operational readiness of a software package (JAVS). Therefore, an important aspect of the COMPOSE testing is the evaluation of errors found in JAVS during the tests to determine why they were missed during JAVS self-test.

### 3.1 PRELIMINARY JAVS EVALUATION FROM SELF-TEST

The basic data driving JAVS during development was derived from a variety of sources, including some inputs that were generated solely for the purpose of stressing the analysis algorithms. An extensive series of tests of JOVIAL language coverage was derived from the JOVIAL Compiler Verification System furnished by RADC; other tests came from the JOVIAL Compiler Tests furnished by System Development Corporation. A subset of these development tests was used to provide the self test input data.

The self test consisted of the following procedures:

- Process each of JAVS's twelve functional components by the syntax and structural analyzers
- Instrument each component separately
- Obtain JAVS documentation reports on each component
- Execute JAVS in the normal manner except to load one instrumented component with the remaining uninstrumented ones in each run
- Obtain JAVS post-test analysis reports for each component

The goal was to exercise 85% of all source statements; in achieving this goal, 70% of all DD-paths were executed.

Because the test case data had been designed during the program development, and because of the structure and design of JAVS, only one test case was needed for testing all but one of the components (the syntax analyzer component which required an additional test case). The self-test demonstrated the following:

- Applying JAVS to a large test object (JAVS itself) is feasible.
- Well-structured software, although large, lends itself well to AVS test techniques.
- Software implemented to reflect functional requirements achieves high coverage when the test data covers the functions.
- High quality documentation produced by JAVS is invaluable in both testing and software maintenance.
- Test case generation is best generated during development of the software.
- The development team's knowledge of the design of the test object, operation of the test object, and experience in using JAVS, contributes immeasurably to the success and efficiency of the testing activity, and coverage requirements (i.e., a specified level of statement or DD-path testedness) offset the natural tendency of the development team to test subjectively.

The full effectiveness of this structural testing could not be assured until the JAVS was stressed with large amounts of complex source code. This additional stress was supplied by testing COMPOSE.

## 3.2 JAVS EVALUATION FROM COMPOSE TESTING

Section 2 described the COMPOSE program and the JAVS-supplied tests of it. This section discusses the problems uncovered in JAVS as a result of testing such a large and complex program. The basic problems encountered fell into five categories:

1. Residual errors in JAVS
2. Weaknesses in the JAVS syntax analyzer
3. Incompatibility between the syntax analyzer and the JOCIT compiler
4. Size limitations and language constraints in JAVS
5. Compiler errors in compiling JAVS software.

On the positive side, it is clear that the test team was able to do effective testing with meager knowledge of COMPOSE. Effective testing with the available resources would not have been possible without JAVS, and in the absence of a measure of effectiveness there would have been no indication of the extent of testing that had been achieved. The effort also greatly improved the Air Force documentation of COMPOSE with up-to-date enhanced source listing and JAVS documentation reports, in addition to the JAVS testing reports.

One indication of the value of testing experience coupled with JAVS assistance is provided by the subsequent test at SAC Headquarters. At the end of the RADC tests the updated JAVS was installed at SAC under the WMMCCS operating system. Drawing on RADC experience with COMPOSE, the team was able to complete a significant testing effort of another COMPOSE component in only three days.

The component of COMPOSE analyzed at SAC Headquarters comprises 72 modules (3,200 JOVIAL statements). Although the computer turnaround time was long, testing this component required only three days and six computer runs. This test benefitted from the test team's knowledge of JAVS utilization, the operation of FMIS and its required input data, and familiarity with the source code. In addition, the JAVS macro commands (see the JAVS User's Guide for macro command description) and the job control cards supplied made processing very straightforward. The source code was processed by the JAVS syntax and structural analyzers, instrumented, compiled in instrumented form, loaded with the rest of FMIS, and executed with a SAC-supplied set of test data. The resulting execution trace file was processed by the JAVS post-test analyzer. The test data exercised 52% of the component's DD-paths (very high coverage in a single test, considering the complexity and error protective code of the test object), and produced a set of JAVS computer documentation reports for the component.

While retesting this component at SAC was not done, analysis of the JAVS execution coverage reports and the JAVS computer documentation of the software by a tester familiar with this component would make additional test data relatively easy to generate.

### 3.2.1 JAVS Errors

Careful account of errors encountered during JAVS operation was made to determine their origin and provide insight into error detection analysis and practical advancements to AVS tools. Using the first release of JAVS at RADC as a control version (i.e., enhancements made to JAVS during the current contract were not considered), ten errors were detected during JAVS's extensive operation with COMPOSE. These errors fall into three major categories: structural, design, and logic errors.

One of three structural errors was detected during the self-test. Another error manifested itself during the self-test but the output was not detected as erroneous. The third structural error was simply one of the untested (but important) DD-paths. More thorough DD-path testing would have uncovered this infinite loop error.

The four design errors were in the syntax analyzer which received the lowest statement coverage during the JAVS self-test. The syntax analyzer was not designed for JAVS specifically but was adapted for JAVS utilization. Three of the errors were detected as the result of processing uncommon syntactical constructs. Nevertheless, the lesson learned in evaluting JAVS is that extra emphasis should be placed on the design phase of critical parts of the software and that functional test data should be designed at the same time.

Each of the three logic errors dealt with a boundary condition and could be detected by data flow analysis or by JAVS EXPECT computation directive. In the errors encountered, if the programmer or tester had had the foresight to check the offending array and tables for their boundaries, the proper program logic would just as easily have been incorporated into the original code. It seems that data flow analysis, in which boundaries are checked automatically, would be an appropriate addition as an AVS tool. Using the current version of JAVS for data flow analysis would be a simple matter if JAVS computation directives are incorporated into the software as it is being coded. Effective usage of the directives in the testing phase requires a thorough understanding of the software and its function.

During the current contract, a number of modifications were made to the JAVS source code. These changes took place as the need became apparent, and the affected JAVS components were not self-tested. Several errors cropped up as enhancements to JAVS were made. The most notable cause for error was the modification to deal with external labels, i.e., labels which are defined in one module and referenced in another*. After the code modifications, five new errors introduced by the changes were identified. In addition, two JOCIT compiler errors were found while processing COMPOSE through JAVS, requiring modifications to JAVS source code. The complete details of errors detected are described in Appendix B.

Conclusions from extensive testing of JAVS are that systematic testing using a well-defined test plan and an automated tool reduced the number of errors detected during subsequent operation to <u>0.03% of the number of statements</u> (JAVS contains 28,239 statements and 366 modules). The errors could have been reduced even further with more rigorous boundary condition decisions in the code and a better design of the syntax analyzer.

---

*JOCIT allows this transfer under certain scope conditions. JAVS lists the external label transfer as a prohibition constraint to performing structural analysis. The construct is not well-documented, and is considered to reflect a poor programming practice. This constraint was ignored after modifications to JAVS were made. The current JAVS syntax analyzer requires that if a label is global (external) it must be defined before it is referenced in a transfer. This may require modification to the user's source code before JAVS processing. Although external labels can be processed by JAVS, their usage is discouraged, and the transfer to a label external to a module is still listed as a prohibition constraint in the <u>JAVS Reference Manual</u>.

### 3.2.2 JAVS Improvements

It was evident to the testers of COMPOSE that JAVS demonstrated effectiveness in automated structural testing and documentation of software. The following subsections describe areas for improvement in making JAVS more generalized and efficient.

Syntax Incompatabilities. The code for the JAVS syntax analyzer was adapted from an existing compiler rather than developed directly for JAVS purposes. This approach was taken to minimize development time. All incompatibilities between the syntax analyzer and the JOCIT compiler, as well as the excessive memory requirements of the syntax analyzer can be attributed to this decision. In spite of extensive language coverage tests, COMPOSE software contained constructs not present in those tests; e.g., JOVIAL keywords in comments, a CLOSE as the first module in a compilation unit, etc. Redesign and rewrite of the syntax analyzer should be the first priority for JAVS modifications.

JAVS Size Limitations. The size limitations of JAVS software fall into two categories: those imposed by array sizes which may be relaxed by recompiling JAVS software, and design limitations of JAVS's Data Management System Component (250 modules on one library) which may be relaxed by an alternative data management subsystem, which should also be designed for greater operational efficiency. COMPOSE software proved to be larger than provided for in both categories. Size restrictions for key arrays were relaxed, and COMPOSE was partitioned into several libraries for testing. The partitioning also affected JAVS operational performance by reducing processing time significantly on analyses which involve complete libraries (e.g., module interdependence). This can be attributed to the volume of data it is necessary to access when looking for matching symbols; i.e., processing requirements related to access to data retained on auxiliary storage can, for large data bases, be significant for searches and sorts. If JAVS were redesigned for highly efficient searching, it could handle larger codes with no increase in processing time over the present version.

Language Usage Constraints. In keeping with the philosophy of treating an invokable module as a separate unit of code for analysis, the JAVS structural algorithms and test analysis place restrictions on transfer of control to other modules: all references to lower level modules are by invocation only, and all transfers to higher level modules are by module return. This restriction is violated by COMPOSE software. JAVS structural analysis was modified to accept other forms of control transfer, but the current test analysis algorithms cannot properly account for these constructs (see Appendix B). Ideally this type of intermodule control transfer should be prohibited by the compiler, but since it is not, JAVS should be modified to identify the transfers, check them for potential errors, and instrument them for proper reporting in the test analysis component.

### 3.2.3 JOCIT Compiler Problem

Some of the problems in COMPOSE testing were traced to the JOCIT compiler. During JAVS installation at RADC, a few errors attributable to compiler malfunction were revealed, mostly in character string comparisons. Since JAVS software deals primarily with source text analysis, string comparison pervades the system, thus it is very vulnerable to this particular compiler malfunction. Rather than modify all of JAVS software for this known compiler problem, alternative source code was supplied to circumvent the compiler malfunction in known trouble spots pending release of a revised compiler. COMPOSE testing revealed additional problems of the same nature that also had to be exceeded in one case.

COMPOSE software approaches size limitations of the JOCIT compiler for COMPOOL and external symbol references. JAVS instrumentation of COMPOSE caused the limitation to be exceeded.

## 3.3 EVALUATION SUMMARY

- The design of JAVS software and its systematic testing resulted in few detected errors in initial operational usage.
- Although designed and developed as an experimental tool, JAVS has demonstrated operational effectiveness on large software systems.
- As a testing tool to measure coverage and capture program flow, JAVS is valuable for both small and large programs.
- As a documentation tool JAVS automatically produces high quality, accurate software documentation invaluable in both software maintenance and testing.
- JAVS operational efficiency can be improved and its module capacity increased by suitable modification to its data management subsystem.
- Some incompatibilities with the JOCIT compiler and other language constraints can be alleviated by replacing the JAVS syntax analyzer and augmenting the structural and testing analysis algorithms.

## 4. SOFTWARE TESTING METHODOLOGY

### 4.1 OVERVIEW

How a particular testing activity is to be accomplished is determined by answering these four very specific questions:

1. What are the characteristics of the software test object? (Sec. 4.1.1)
2. What are the available resources (e.g., hardware, support software, personnel, time, test tools)? (Sec. 4.1.2)
3. What are the test goals? (Sec. 4.1.3)
4. What procedure will be used to accomplish the goals? (Sec. 4.1.4)

There is no general approach that applies to all testing. Each particular testing activity is unique, and should be analyzed to determine suitable procedures. This suggests that the testing process itself consists of three distinct phases: (1) identifying the elements of the test activity, (2) preparing a test plan, and (3) carrying out the planned tests. The remainder of this section addresses the problem of practical application of the testing methodology.

#### 4.1.1 Software Test Object

The software to be tested is called the test object. The characteristics of the software important to AVS-supported testing are listed in Table 4.1. JAVS capabilities and processing techniques pertinent to software characteristics are also shown.

Source Language

JAVS deals with the software test object in source language form. This is the form most suitable for AVS-supported testing because the information required by the compiler is also required by the AVS. JAVS assumes that the source text compiles without syntax-related errors.

Overall Size

JAVS supports testing of both small and large test objects. The software may consist of a single module or it may be hundreds of modules in size. The total source may range from a few statements to tens of thousands. It may consist of one or more compilation units. The limitations of a JAVS implementation on a particular computer are restricted by direct access memory size and the size of auxiliary files.

Organization

The test object may be a complete or an incomplete program; it may be organized into one or more subsystems, or it may be a utility package. If the

TABLE 4.1

CHARACTERISTICS OF SOFTWARE TEST OBJECT

| | Software Characteristics | JAVS Capabilities |
|---|---|---|
| 1. | Source Language | JOVIAL/J3 (may have imbedded assembly code) |
| 2. | Size | Small to large |
| | • Number of modules | 1 to 250 per library |
| | • Number of statements | Unlimited |
| | • Compilation units | Single or multiple |
| 3. | Organization | |
| | • Complete/incomplete program | One or more START-TERMS |
| | • Software partitioning | Linked or non-linked |
| | • Design | Assertion statements, automated documentation, dynamic test identification, modular, structured for best results |
| 4. | Computer System | HIS 6180, CDC 6400, HIS 6080 |
| 5. | Support Software | |
| | • Operating system | WWMCCS for HIS 6080<br>GCOS for HIS 6180, GOLETA for CDC 6400 |
| | • Communications | |
| | • Compiler/assembler | JOCIT JOVIAL |
| | • Link loader | System loader |
| | • Library | JAVS probe routines, sequential file I/O |
| 6. | Suitability for Testing | |
| | • Design | Non-recursive, non-concurrent, non-time dependent, direct correspondence to functional specifications, traceability of symbols to input, identifiable I/O |
| | • Partitioning | Functionally similar components |
| | • Data | Initial test case |

program is incomplete, some additional software must be supplied for test execution. The additional software may be either operational software or test software which, when combined with the test object, results in a complete program. Although the additional software need not be processed by JAVS in source form, the COMPOSE testing experience disclosed significant difficulty due to the absence of that source. At the minimum, the interfaces to the test object must be clearly and unambiguously defined in order to prepare test data.

For a very large software system consisting of many hundreds of modules, it is wise to partition it into test objects of tractable size. Normally, very large software systems are designed as subsystems according to some functional criteria. If the subsystems are each a collection of hierarchically structured, interrelated components or modules, this same partitioning may also be suitable for testing purposes. Miscellaneous collections of low-level utility modules which are invoked throughout the remainder of the system can be grouped together as a separate subsystem.

In partitioning the test object, some consideration should also be given to the resources required to test the partition. The instrumented software requires more main memory from code expansion of the software instrumentation probes and more computer time due to the overhead of executing the probes.

A further consideration in partitioning is the execution-time behavior of the test object. During test execution it is important to identify important events that can separate the test execution into a sequence of individual tests. This permits the tester to extract more information from the test results. Candidate events include the start of an initialization or termination process, the start or conclusion of a new test case, the change in mode of behavior (e.g., from normal mode to error mode), opening or closing of files, memory link loading, and invocation of other software not being tested.

To properly partition large test objects the tester must use documentation supplied with the software for guidance. The supplied documentation, which may be manually prepared, may not correspond exactly to the test object, since more often than not this type of documentation does not reflect the current version of the software. JAVS documentation capabilities offer automated assistance in verifying the accuracy of the supplied software documentation. For example, the intermodule dependence reports give a concise picture of the interface of one set of modules to all referenced modules. Trial partitions of the software may be defined by the user, and verified with these reports from the JAVS. If the software is too large to be processed by the JAVS as a single unit, initial partitioning must be based on supplied documentation.

## Computer System

Under normal operational conditions the test object executes in a host computer system. This system may also be suitable for testing. By using an intermediate test file to record the results of test execution, only the JAVS data collection routines directly interact with the software test object during test execution.

The computer used during test execution must have sufficient excess resources to accommodate the code expansion due to instrumentation and recording of test probe data. JAVS may, or may not, execute on the same host computer as the test object. JAVS requires that only the data collection routines reside on the same computer as the test object.

## Support Software

The test object may make use of a variety of support software which is not normally considered part of the application. For example, it may execute under control of an operating system with or without support of communications software (i.e., as a time-sharing application). A compiler (or assembler) is essential to translate source text into object text. Furthermore, support by a link loader and access to required library routines are also necessary. For JAVS-supported testing, the same support software is used.

## Suitability for Testing

There are additional software characteristics which affect its suitability to JAVS-supported testing. These are a result of assumptions made in the JAVS implementation itself. For example, the current JAVS implementation assumes the software test object contains no recursion, has no concurrent paths during execution, and is not time-critical. Each of these restrictions may be relaxed in future JAVS implementations.

Some software design characteristics facilitate JAVS-supported testing. Among these are:

- Highly modular, structured code
- Direct correspondence of implemented software to functional specifications
- Localization of code controlling important events in software behavior (e.g., new test case, file I/O, link loading)
- Identifiable module inputs and outputs
- Mnemonic symbol names
- Traceability of symbols to input symbols

Existing software may fortuitously possess some, if not all, of these properties. JAVS-supported testing is hampered if the software test object lacks these, or if it contains logically unreachable code, uses borrowed code, or by-passes normal module invocation and return protocol for control transfer.

Software may also be deliberately designed to take advantage of specific JAVS capabilities such as use of imbedded assertion statements for dynamic checking of expected behavior, automated documentation, or program performance with test point identification.

### 4.1.2 Test Resources

The resources available for testing must be identified before an approach to testing can be determined. These include the computer resources, data for executing the software, the members of the test team, the JAVS capabilities, and the time frame within which testing must be completed.

#### Computer Resources

All computer resources (both hardware and support software) used by the test object during normal execution should be identified. This information is usually contained in software documentation or can be extracted from sample execution runs. For example, for programs which execute under control of an operating system, the hardware and software requirements may be gleaned from reports produced by the system loader and information extracted from job control statements. If the program is overlayed (i.e., different parts of the program reside in main storage at different times), then the memory layout of each overlay link is also needed. This mapping of the test software onto the computer is referred to as the execution environment of the program.

Test Execution replicates normal execution in the sense that the program reads its own inputs and produces its own outputs. Additional output is captured from the instrumented program during Test Execution for later analysis by the JAVS. Test Execution differs from normal execution in the following ways:

- Some or all of the program has been instrumented to capture execution behavior on a test file.
- Execution output will include the results of any assertion statements embedded in the source code.
- The instrumented code, although logically equivalent to the uninstrumented code with probes added, contains references to JAVS probe routines which are added to the load sequence.
- The probe test file is recorded.

The major effects from these software perturbations are the increased memory requirements and execution time due to code instrumentation.

In some instances the additional computer resources required preclude testing in the normal execution environment. Other considerations, such as accessibility and operational compatibility with the JAVS execution environment, may also indicate a change in the execution environment for Test Execution. For example, the FMIS software (see Sec. 2) normally operates under time-sharing mode in the GCOS operating system. Expanded core requirements of instrumented code, together with need for rapid analysis of test file data by JAVS, made it more practical to test FMIS as a series of multiple-activity batch jobs. In this mode, the first job uses JAVS to instrument the desired component, the second compiles the instrumented code, and the third loads the test, executes the (partially) instrumented FMIS, and uses JAVS to analyze the test results.

Suitable modification to operational procedures must be made to accommodate the changes made for testing. For programs which execute under control of an operating system, these modifications include altering the job control language to provide for compiling instrumented code and merging it with non-instrumented code, loading the JAVS probe routines, and capturing the test file for analysis by the JAVS.

### Data Requirements

For Test Execution, the software is exercised with test data in the test environment. This data may be generated specifically for the JAVS-supported test activity, or it may be taken from previous execution of the software, or both. Existing test data, as well as results from tests unsupported by JAVS, can be invaluable to the test process, especially if the existing data is coupled to functional requirements of the software. For example, FMIS uses two types of input data: user commands and a complex data base. In JAVS testing, the data found to have the most direct affect on FMIS program coverage was the user command data, with the data base of lesser importance. Testing was conducted with an existing data base, holding the data base input fixed and altering only the user command data.

### Test Team

Testing requires that the test team be capable of preparing data, making computer runs with the software, and analyzing output produced during execution tests. In addition, JAVS-supported testing requires that the test team know how to (1) use JAVS capabilities for analyzing the software, (2) make suitable modifications for test execution, and (3) analyze the combined test results from the software and JAVS (i.e., normal output from the software together with JAVS post-test analysis of coverage derived from software probes). It is important that the test team have more than superficial knowledge about the test object. In particular, the team must have specification-level knowledge of functional behavior, at least limited knowledge of program structure, and detailed knowledge of operational requirements.

For effective use of JAVS, the test team must understand the purpose of each of the JAVS processing capabilities and select whatever capability is applicable at each stage during the particular test activity. JAVS can be used to accelerate testing. For example, if the test team's knowledge about the software structure is deficient due to lack of detailed documentation, then before actual testing the JAVS documentation capability can generate the detailed information about inter-module dependencies derived directly from the software. The additional information needed about each module to give meaning to the invocation structure includes each module's purpose, its inputs and outputs, and the interpretation associated with data processed by the module.

## 4.2 TEST GOALS

The overall testing goals are to improve the quality of software through testing and validation of test results. Detailed testing goals are directly related to the type of testing to be done: single module testing or system-wide testing. The type of testing, in turn, may depend on the stage of software development and previous test history of the test object. Often single module testing is most appropriate during the code development phase or whenever a module has been changed or replaced during the maintenance phase. System-wide testing is applicable whenever collections of modules are tested (e.g., in software integration and maintenance phases, in top-down development, or in acceptance tests).

### Single-Module Testing

For single-module testing, the testing process for complete coverage has a single objective: to construct a usefully small set of test cases which, in aggregate, cause execution of each DD-path in the module at least once. Real programs may have DD-paths which cannot be exercised, no matter what input values are used. In that case, an explanation of why a path cannot be executed should be provided. Programs which have been tested to this level will meet the following criteria:

- Each statement in the program will have been executed at least once.
- Each decision in the program will have been brought to each of its possible outcomes at least once, although not necessarily in every possible combination.
- Each unexercised DD-path will have been analyzed by the tester.

### System-Wide Testing

For system-wide testing the testing goal is a straightforward extension of the single-module testing goal: to construct a usefully small set of test-cases which exercise as many DD-paths as possible, out of the aggregate set of DD-paths in all modules in the system. The coverage measure may be an overall percentage of DD-paths exercised, the percentage of DD-paths exercised in the least-tested module, or the percentage of DD-paths exercised for the least-tested subsystem of modules. A more stringent test goal would measure coverage in relation to module location on the invocation hierarchy. For example, any modules which are referenced by more than one other module might have their coverage separately determined for each referencing module.

## 4.3 TESTING STRATEGY

In previous subsections the discussions have focused on defining the test object and collecting information about its structure, operation, and the effects of using JAVS in testing. This subsection presents a general methodology for testing a software system and gives guidelines for effective JAVS usage. Detailed information about JAVS usage is contained in Refs. 2,3. The sequence of major steps for testing with JAVS are:

1. Describe software behavior
    - Understand software-system functional requirements

- Generalize modes of system behavior
- Define system hierarchical structure

2. Develop test plans keyed to modes of behavior and software structure
3. Execute functional tests
4. Develop structure-based tests for increased coverage

## 4.3.1 Describe Software Behavior

### Functional Requirements

The top-level of understanding software behavior is the functional requirements level. If the software is adequately documented, there will be a clear statement of functional requirements that can be easily related to the modes of behavior described next. If such documentation is not available it is important to spend some time developing such a functional description. Not only is it necessary for understanding the modes of behavior, but it is also mandatory for the development of meaningful functional tests.

### System Behavior

*The next level of functional description* is called modes of behavior. In general, there is more than one mode of behavior associated with each functional requirement. For example, a data base management system may have a primary, high-priority mode which handles user commands interactively, and a secondary, background mode which generates periodic reports on system usage. Other modes of system behavior may include error processing or operation under degraded conditions (e.g., with an incomplete or garbled data base). If there is more than one mode of expected behavior, each should be identified and named.

### System Structure

Having defined the system modes, an attempt should be made to relate each mode to a hierarchical structure of the program. There are several kinds of structure in a computer program: data structure is the organization of the data on which the program operates; computation structure describes the program's operations on the data; and control structure is the means of organizing the computations in the software. The control structure is dealt with directly by the JAVS-supported testing methodology. The other kinds of structure are tested only insofar as they interact with the control structure.

Large software systems are usually organized into a series of subsystems, subsystems into components, and components into modules. This static organization describes the way the individual elements of the system depend on one another, without regard to the way program control flows up and down. The dynamic organization of a software system is the structure which results from considering the effects of all invocations between modules, components and subsystems. It is usually called the invocation hierarchy of the software system. The testing methodology deals with the dynamic organization. Normally, program documentation will identify the static organization of software modules. The dynamic organization can be derived from more detailed documentation and verified with JAVS documentation capabilities showing intermodule dependencies.

For each named mode of behavior, a software description should be developed which identifies the modules invoked, the specified input domain of each module, and the expected system performance. Existing software documentation may not contain sufficient information to document each behavior mode separately. It may be necessary to execute a limited number of functional tests with the software instrumented at the module invocation level to determine which modules are invoked and under what conditions.

## 4.3.2 Test Plans

To minimize testing effort, test plans should be developed which are keyed to the named modes of behavior and to software structure. Each test plan should identify one or more functional tests for initial testing, and the software structures to be tested. Often more than one substructure of the software system will be exercised with a given test case. This (so-called) collateral testing can greatly reduce the overall testing effort. Each test plan should contain the following information for each functional test:

- A name identifying the test
- What function is tested
- The primary code structure tested
- Collateral code structures tested
- A description of the resources required
- The expected performance
- Criteria for evaluating the test

All of these items are commonly called for in test plans for software acceptance tests. Wherever appropriate, use should be made of existing functional tests and test plans. The major distinction between testing with and without JAVS is that with JAVS there is an orderly progression of testing from the initial tests through well-defined steps to achieve the desired testing coverage in addition to satisfying the test criteria. Quite often, additional tests to achieve increased coverage are derived from the initial functional tests.

## 4.3.3 Functional Tests

Before processing with JAVS, each of the initial functional tests should be used to exercise the software and the output evaluated against the test criteria. This is important for two reasons:

- It provides a baseline set of output from the uninstrumented software for purposes of later comparison to that from instrumented software.

- It demonstrates the ability of the test team to prepare test data, execute the program, and interpret results.

This step requires that the program be complete, that source code be compiled error-free, and that test data result in acceptable execution (though not necessarily expected behavior).

The next step is to execute each of the functional tests with instrumented software and determine initial coverage. This necessitates JAVS processing to build the library (BASIC and STRUCTURAL), instrument appropriate portions of the software (INSTRUMENT), compile and execute the instrumented software with the JAVS probe routines (Test Execution), and obtain coverage reports (ANALYZER). It is very important that normal program output from Test Execution be checked against the baseline output. If discrepancies exist between the two, this is direct evidence that the addition of instrumentation has, in some way, exposed a software malfunction. Some of the reasons for this type of error are:

- The test object is sensitive to time or space perturbations (i.e., it is not suitable for JAVS testing).

- The addition of probe routines was not properly accomplished (e.g., placed in the wrong overlay link).

- The test data or test environment is different from that of the baseline test.

- The compilation process has caused the malfunction (e.g., using the wrong COMPOOL to compile; inconsistent code generated by the compiler).

- Computer resources are inadequate to process instrumented code (e.g., compiler limitations regarding number of external symbols or memory capacity for expanded code).

JAVS capability for evaluating test effectiveness (ANALYZER) provides a detailed and comprehensive analysis of testing coverage. Reports on execution tracing, module and path coverage, timing, and modules and paths not exercised are generated. For the initial functional tests this information should be evaluated in some detail since it represents the point of departure for subsequent testing. Since JAVS only assists the test team in preparing the software for Test Execution, the initial coverage results may not accurately reflect actual coverage. For example, JAVS requires the tester to select the placement of a "beginning of test" signal and an "end of test" signal to the probe routines and they in turn record coverage only during the user-selected interval of execution. Thus, coverage reports are limited to code executed within this interval. Some reasons for unexpected coverage results are:

- The functional test does not exercise the expected modules at all.
- The selection of test point placement is improper.
- The selection of modules to instrument is not compatible with the functional test, perhaps indicating erroneous definition of system substructure.

If anomalous results occur at this point, it is important to review decisions made during the previous steps in the testing process before proceeding.

### 4.3.4 Structural Tests

Once the test team is confident of the quality achieved in obtaining initial functional test results, testing proceeds with JAVS assistance as follows:

- Selecting a testing target from information in JAVS coverage reports for previous tests
- Constructing new tests to improve coverage using JAVS retesting assistance
- Performing Test Execution with new tests capturing software behavior data
- Analyzing test results from JAVS coverage reports

These steps are repeated until the test coverage objectives have been met.

Software testing can be performed at the single-module or system level. At the single-module level, the retesting target is a set of DD-paths which have not been exercised. Retesting at the system level involves identification of target modules with low coverage and analysis of intermodule dependencies.

In order to construct new test cases, information about the control structure of the module and its input domain is used. The relationship of the module's input domain to the system input data must be determined in order to test the module in its normal environment (i.e., its position in the invocation structure of the software system). This may prove difficult, or not at all possible, since communication paths to the module may be blocked (e.g., by protective code or by lack of knowledge as in FMIS testing). If this is the case, a special test environment may be needed to thoroughly exercise the module.

#### Systematic Single-Module Testing

The testing process for single-module coverage has a single objective: to construct test cases which cause execution of as yet unexecuted DD-paths within the program. Testing is over when all DD-paths have been exercised or when those which have not been exercised are shown by the program tester to be logically unexecutable.

Two questions arise in addressing the task of single module retesting: What are the targets for retesting? and, once selected, What assistance is available to exercise the targets?

The DD-path selection criteria should attempt to maximize collateral testing; i.e., exercising more than one unexercised decision path with each new test case. Several guidelines can be used to aid the selection:

1. In a cluster of unexercised paths, choose an untested DD-path that is one of the highest possible control nesting level. This selection assures a high degree of collateral testing, since some of the DD-paths leading to and from the target must be executed.

2. A "reaching set" is a sequence of DD-paths which lead to a specified decision path. At user request, JAVS determines reaching sets which include or exclude iteration. As the DD-path target choose a DD-path which is at the end of a long reaching set. In addition to collateral testing, it is likely that the resulting test case input will be similar to data which resembles the functional nature of the program.

3. If a prior test case carries the program near one of the untested DD-paths, it may be more economical to determine how the test case can be modified to execute the unexercised path.

4. If the analyses required for a particular DD-path selection are difficult, then choose a path which lies along the lower level portions of its reaching set. This can simplify the analysis problem.

5. Analyze the untested DD-path predicate (conditional formula) in the reaching set for "key" variable names which may lead directly to the input.

6. Choose DD-paths whose predicates evaluate functional boundaries or extreme conditions; exercising these paths frequently uncovers program errors.

Most of these guidelines depend upon the identification of DD-path reaching sets. One of the tasks performed by the JAVS testing assistance processor (ASSIST) is the determination of these sets. The user inputs the desired path number to be reached, and ASSIST generates the reaching set of paths from the module entry or from a designated starting path to the specified path. For the generated set of paths, the key program statements are printed, including the necessary outcome of any conditional statements which are essential. The user may specify iterative or non-iterative reaching sets to be generated.

The process of relating paths which are targets for retesting to the input for generating new test cases is highly dependent upon the design of the test program. JAVS shows what code segments have not been exercised by the data and the program paths that lead up to any selected DD-path target. The tester must analyze the predicates in the testing targets for important variables which may be described in program comments or documentation. These variables can be traced throughout the program by using the JAVS cross reference report and the module interdependency and invocation parameter reports (if the variables are passed as parameters).

When the new set of test cases is generated, it can be added to the previously input data or executed alone by the instrumented modules. The results of this Test Execution are then processed by the post-test analyzer to see if the coverage is satisfactory. At this time, the user may see problems in the software which require code changes. The JAVS documentation reports can be used to determine the effects within a single module or within the data base system of modules that the code modification will have.

## Systematic System-Wide Testing

The system testing effort can be organized according to two fundamentally distinct strategies: (1) bottom-up system testing, and (2) top-down system testing.

Bottom-up Testing: This testing strategy attempts to provide comprehensive system testing coverage by building test cases from the bottom of the system invocation hierarchy first, and extending these test cases upward during the continuing and concluding testing phases. Bottom-up testing may require the use of special testing environments (see below), but is likely to achieve the best overall testing coverage.

Top-Down Testing: This testing strategy deals with an entire software system first, and, after subsytem (or component) testedness is measured, proceeds downward through the software system's invocation structure. Test case data is added only at the topmost level and, as a result, a set of systemwide test cases are developed directly.

The optimum system testing strategy for a particular system generally combines the two strategies. The choice is based on the level of coverage achieved, the difficulty of proceeding upward or downward in the system organization, and the effort required to establish a testing environment in each case.

The basic ingredients of the systematic software system testing methodology are the following:

- The ability to perform comprehensive single-module testing for each invokable module
- Knowledge of the system's invocation structure
- Previous (and initial) system testing coverage measures
- A next-testing-target selection function to allocate testing effort.

The general form of the system testing methodology is shown in Fig. 4.1, which emphasizes continuous use of a system testing coverage measure. The interaction between the system testing coverage measure and the process of selective application of the single-module testing methodology is described next. The coverage value can also be used to select the best next-testing target.

System-wide testing coverage can be measured in terms of the coverage for each module, or in terms of the coverage for an identifiable subset of related modules (i.e., a component). The coverage measure can be used to select the best next-testing target.

The simple per-module coverage measure will direct testing effort toward the module which is the least tested. The per element form of composite testing coverage allocates testing effort toward the component which has undergone the least testing.

The measure actually used should depend on the internal structure, and possibly the functional requirements, of the software system as a whole. The measure should unambiguously identify the module(s) least tested, but should tend to identify a number of possible testing targets. The choice between them should be made within the confines of the invocation hierarchy, and by considering the two important variations of testing strategy: top-down testing, and bottom-up testing.

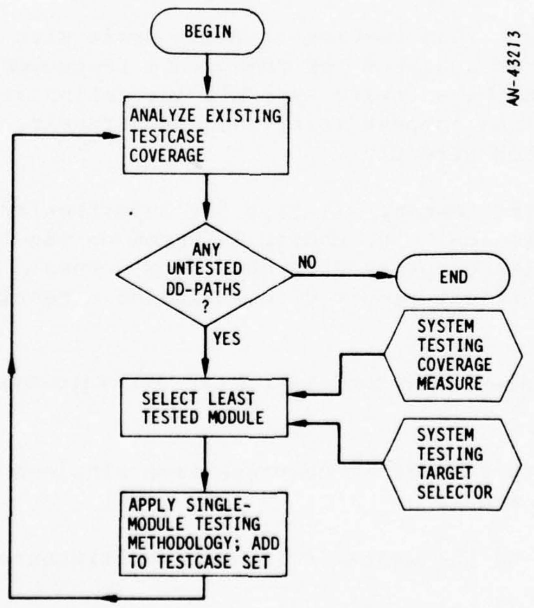

AN-43213
BEGIN
ANALYZE EXISTING TESTCASE COVERAGE
ANY UNTESTED DD-PATHS ?
NO
END
YES
SYSTEM TESTING COVERAGE MEASURE
SELECT LEAST TESTED MODULE
SYSTEM TESTING TARGET SELECTOR
APPLY SINGLE-MODULE TESTING METHODOLOGY; ADD TO TESTCASE SET

Figure 4.1. Overview of System Testing Methodology

## Bottom-Up Testing

In bottom-up testing, a system tester has two choices: where to concentrate the testing effort, and where to provide testcase data.

1. Testcase data can be supplied through the existing data input points. The system tester must be aware of data transformations performed prior to delivery to the module on which he is currently working. These transformations may make it difficult, or impossible, to exercise the current testing target.

2. The testcase data can be supplied through a separate testing environment designed and implemented specifically to provide for testing of a single system element. At the system testing level the testing would be performed with the normal data input mechanism.

The choice between these two mechanisms for providing testcase data must be based on the specific internal features of the software system.

JAVS intermodule dependencies and symbol cross reference capabilities can be used along with module input and output information to select the appropriate technique.

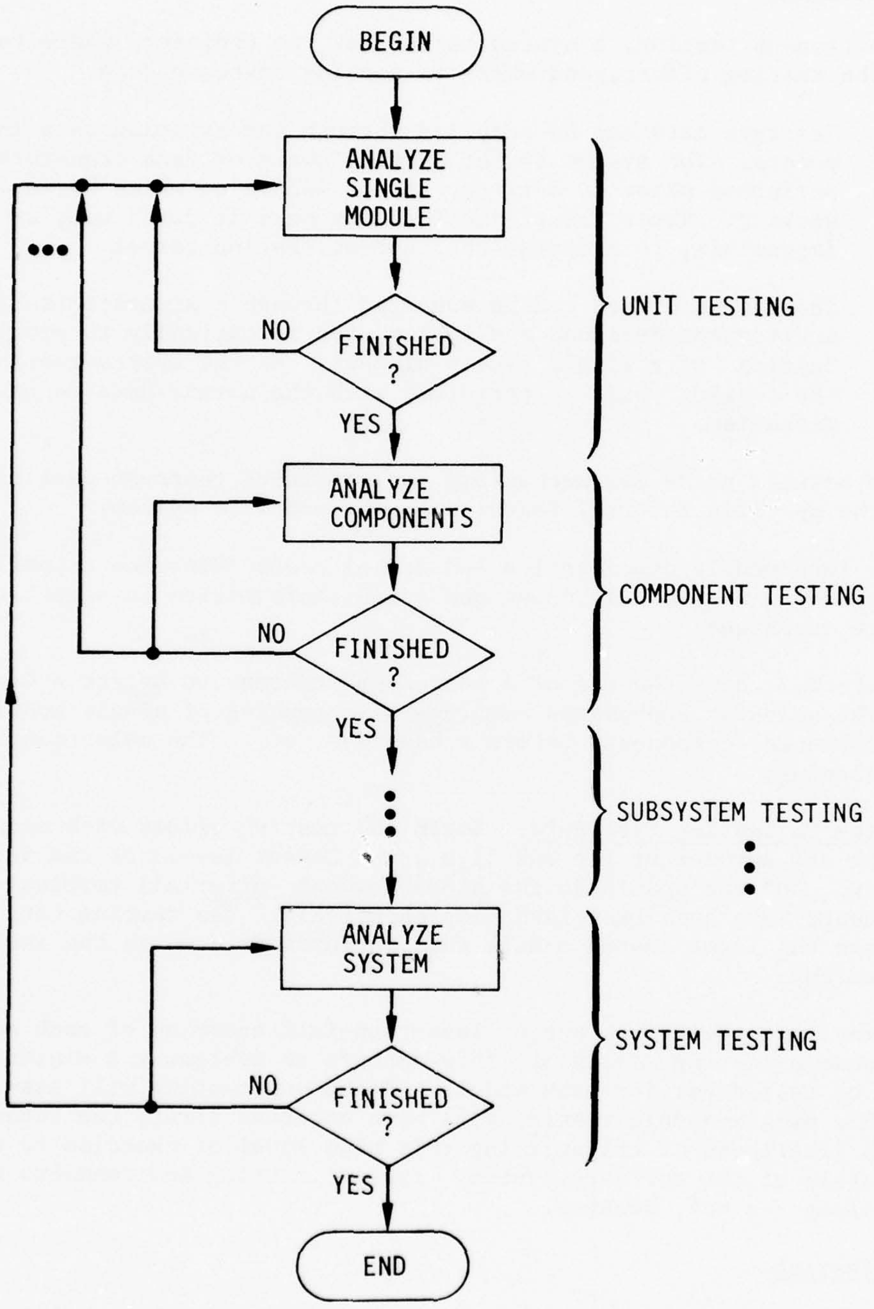
Figure 4.2 shows the use of a bottom-up strategy to select a testing target. The selector emphasizes comprehensive testing of single modules before components, components before subsystems, etc. The selection rule is the following:

> Bottom-Up Testing Procedure. Begin the testing effort with modules which are invoked at the end (i.e., the lowest level) of the invocation chain. Advance upward in the hierarchy only after all terminal-branch elements have been exercised comprehensively. The testing target is always the least tested module that is furthest down in the invocation hierarchy.

It may be necessary to accept less-than-full exercise of each module as a reasonable testing strategy. This amounts to assigning a minimum threshold of testedness for each module. Bottom-up testing will assure that all possible single-module testing will have occurred first; the technique has a high likelihood of transmitting this high level of exercise to the topmost levels of the software system. Special testing environments may be required along the way, however.

## Top-Down Testing

For top-down system testing, the testing environment at each stage is the obvious one: The topmost element of the software system will control some data and will selectively pass it downward in the invocation structure to the subsystems, to the components, and, eventually, to individual modules.



AN-43214

Figure 4.2. System Testing Methodology (Bottom-Up)

New test case data is added at the points where the software system normally accepts input data. These normal data input points are not necessarily part of the topmost program; there may be special "data entry" subsystems or components which are invoked by the topmost program specifically for this purpose.

Top-down testing is almost the reverse of bottom-up testing, and involves selecting for further testing effort the largest elements of the system (i.e., modules which control entire subsystems or components) before attempting testing of modules at the lower levels. The selection rule that corresponds to this testing strategy is:

> Top-Down Testing Procedure. Begin testing at the beginning of the invocation structure (i.e., at the highest level). Advance downward only after the highest level modules have been comprehensively exercised (to a preset threshold). The testing target selected is always the least tested module which resides at the level in the invocation hierarchy at which testing is currently proceeding.

This selector operates strictly in terms of the chain depth within the invocation structure tree. It is generally undesirable to accept less than 100% testing coverage at any stage of the process since doing so may mean that an important invocation chain is missed. Any invocation chain which is the only one that permits testing some lower-level module must not be skipped in the early levels of testing.

The top-down method generally assures maximum collateral testing prior to attacking any particular lower level module. Stubbing of system elements may be necessary to conserve limited testing resources, however.

## Testing Assessment

At the conclusion of the testing an assessment should be made of the results. This summary should include the following:

- Documentation of the methods and extent of testing: strategy for testing, coverage achieved, test cases used, dynamic behavior modes tested, and identification of logically unreachable code

- Determination of the consistency between the software functional specifications: what specific functions are implemented, what unspecified functions are implemented, what unspecified restrictions are embedded

- Evaluation of existing software documentation: errors, inconsistencies, missing information, superfluous information

## 4.4 GENERAL STRATEGY

The best approach for systematically testing a large software system will depend on the specifics of that system's elements; it is not possible to state a universally applicable strategy. Mixtures of the top-down and bottom-up approaches may well cost the least, and may result in the greatest testing coverage.

The results of a test activity depend to a large extent on the capability and ingenuity of the test team. JAVS does offer tools not previously available to make testing more effective. Application of those tools to particular situations is the responsibility of the testers. There are however, some guidelines for selecting the most appropriate JAVS capabilities for particular situations, and some of these are described below.

Incomplete documentation. Use JAVS resources to build the library (BASIC and STRUCTURAL) and obtain documentation reports on the software (DOCUMENT). Construct missing documentation needed to start testing.

Incomplete software. Use JAVS resources to build the library (BASIC and STRUCTURAL) and obtain documentation reports on the software (DOCUMENT). To complete test environment, first identify top-level modules and external library dependencies. Next, construct a driver for each top-level module: Use JAVS module invocation definition and cross reference for calling protocol and input domain. Provide stubs (i.e., dummy modules) for externals which are referenced but are not present on system or auxiliary libraries: Use JAVS module invocation references for calling protocol. Documentation supplied with the software should be consulted for module interface specifications.

Single-module testing. Use unexercised DD-paths report (ANALYZER,NOTHIT)* to identify potential test target paths. To get the control nesting level of unexercised DD-paths, use the module listing (PRINT,MODULE)* or DD-path definitions report (PRINT,DDPATHS).* Use control flow picture (ASSIST,PICTURE)* for an overview of module structure. Select testing target from reaching set for target path (ASSIST, REACHING SET)* and determine module inputs which cause target path to execute. Generate test data (see below).

System-wide Testing. Use module coverage summary and DD-path coverage summary to identify potential test target modules. For any module never invoked use inter-module dependencies (DOCUMENT) to determine what modules cause it to be invoked directly and indirectly through other modules. Identify which higher level modules were executed. Using cross reference reports together with inter-module dependence reports to identify what modules affect invocation, modify test data to cause invocation of target module. Several cycles of top-down testing may be required if the unexercised inter-module control structure is at all complex. Apply single-module testing techniques to increase intra-module coverage.

---
*See Sec. 5 of the JAVS Reference Manual, under the command headings (Ref. 3).

Unknown behavior. Plant document probes (PROBD) to capture imbedded descriptive information in test execution trace. Examine test trace report to identify behavior with probe location. Use results to select appropriate test points.

Unexpected behavior. Use JAVS assertion statements to isolate causes. At the beginning of the module, insert assertion statements for expected condition of module inputs. At the end of module, where control is returned, insert assertion statements for expected conditions of module outputs. At intermediate locations in the module, insert assertion statements for expected conditions of module behavior. Examine Test Execution output for report of unexpected behavior.

There are several side benefits to be realized from testing. For example, the software can be optimized by removal of unreachable code or code which implements extraneous functions. JAVS documentation reports are useful here in determining the extent of changes to the software (e.g., modules and data structures affected) and the amount of retesting necessary after software modifications have been made.

## 5 ADVANCED AVS CAPABILITIES

Before Automated Verification Systems were available, software developers of necessity relied on extensive manual testing to demonstrate software performance. Information about software characteristics was also largely manually composed, supplemented by meager reports generated as side products from compilers, loaders, and other system software. With access to an AVS with current capabilities (e.g., JAVS and RXVP*) the situation changes dramatically: the software developers can expect to systematically test their software with support from specially designed test tools, automatically document the software of various stages in its development, and build-in quality during code development by utilizing AVS features which perform both static and dynamic tests for software quality. Testing experience with JAVS has demonstrated some direct benefits of AVS usage on existing software not only in achieving comprehensive software testing but also in generating high quality, accurate software documentation. The indirect benefits of these tests included highlighting software properties that are difficult to test, and identifying extraneous code and unused data structures.

More-advanced Automated Verification Systems can be expected not only to automate additional test, documentation, and quality checking services for existing software, but also to improve the quality of new software by other mechanisms such as tool-supported language extensions. To achieve full benefit from the availability of AVS capabilities, new software should be designed to exploit AVS support. Improvements in existing software can be made by taking advantage of maintenance and testing activities to insert additional statements that permit more extensive static analysis by an AVS (see Sec. 5.2.1).

This section presents recommendations of the capabilities to be included in an advanced Automated Verification System. Current AVS capabilities emphasize testing of existing software; future AVS capabilities will include tools for analyzing both existing software and new software which is designed to exploit AVS usage. All of the capabilities are feasible, and many have already been incorporated in GRC's Software Quality Laboratory for the Army.[8,9] In the more distant future, developments in language, independent of AVS considerations, will most certainly influence the direction of advanced AVS development.

### 5.1 CURRENT AVS IMPLEMENTATION

Today's AVS is applicable to today's software. The typical software system subjected to an AVS (1) already exists, (2) is implemented in a popular procedural language such as FORTRAN, JOVIAL, or PASCAL, (3) most likely was not developed with top-down, bottom-up, or structured programming development techniques, (4) is not well-documented for ease in testing and maintenance, and (5) was not designed to exploit AVS capabilities. The currently operational AVS is an experimental tool, incorporates state-of-the-art analysis techniques, operates independently of other software tools (e.g., compilers), and is not yet a primary tool in the software development cycle as are compilers, linkage editors, and operating systems.

---

*RXVP is an automated software verification tool which analyzes FORTRAN, PASCAL and JOVIAL J3B.

To date, AVS usage has been limited and experimental in nature. The objectives of the current effort are primarily concerned with gaining experience in JAVS usage, developing techniques for effective AVS usage, measuring JAVS performance, and defining the extent of applicability of systems with JAVS capability. The limited testing experiences described in Ref. 3 and Sec. 2 of this report used only a subset of existing JAVS capabilities. None of the test objects was designed to use an AVS in testing; furthermore none was implemented with language extensions (i.e., executable assertions) or with the aid of documentation tools provided by JAVS. In addition, the test teams were largely inexperienced in operational AVS usage, although they included individuals who developed several AVS systems. In spite of this, the effectiveness of JAVS in testing and documenting both large and small software systems was demonstrated. For example, the high statement coverage requirements in JAVS software acceptance tests was quickly accomplished and certified with AVS support for instrumentation and test coverage analysis. To do the same task manually on JAVS software would be impractical.

In documenting JAVS software the task of producing the required documentation, to the level of detail called for, was greatly simplified by using the AVS to produce all the structural information. Only the descriptive semantic and organizational details were manually prepared.

JAVS capability is fully described in Refs. 2 and 3. In the following subsections, the capabilities to be found in advanced AVS tools, RXVP and Software Quality Laboratory (QLAB), are described. They fall into two main categories: static tools, which analyze the properties of the software without executing it, and dynamic tools, which are associated with executing the software. Where appropriate, current AVS capabilities are indicated as are those which depend on extensions to current languages.

### 5.1.1 Static Tools

Static AVS tools analyze source code as it is written without executing the compiled code. Some tools are concerned with a small part (e.g., modules) of the software system while others analyze subsystems, (e.g., functionally related sets of modules) or even the entire software system. The categories described are:

- Syntactic Documentation
- Control Structure Analysis
- Consistency Checking
- Program Proof of Correctness

#### Syntactic Documentation

Comprehensive syntactic documentation of software systems is essential to testing and software maintenance activities. The most common forms are enhanced software source listings, symbol tables, and cross reference lists. Typically, AVS tools enhance the software source listings with automatic control structure indentation and nesting level indicators. Symbol tables identify symbols used and symbol specifications (e.g., type, precision, dimensions, scope). Cross reference lists identify where (e.g., by module or by statement) and how (e.g., declared, set, used, invoked) symbols are referenced.

Since some compilers operate on separate compilation units (e.g., the GCOS JOCIT JOVIAL compiler accepts only a single START-TERM sequence), it is not possible to obtain a cross-reference listing over a <u>selected set</u> of modules directly from the compiler. An AVS with access to the complete source of the software can generate this information readily. The designated set of modules may be the complete software system, a functionally related subsystem, those modules resident in a single memory load, those on the same chain in an invocation hierarchy, or other suitable combinations.

In addition to the above, current AVS tools offer more extensive reports such as:

- Explicit module invocation matrix, showing direct invocation of each module to all other modules, whether or not known to the AVS (JAVS, RXVP, QLAB)
- Inter-module invocation tree, showing each module's position in the invocation hierarchy (JAVS, RXVP, QLAB)
- Module invocation environment, showing the local invocation hierarchy of the designated module by reporting other modules which call the designated module and those called by the designated module both directly and indirectly for a specified number of levels (JAVS, RXVP, QLAB)
- Module invocation source text, showing formal invocation declaration of the designated module, all actual invocations of the designated module known to the AVS, and all invocations from the designated module (JAVS, RXVP, QLAB)
- Symbol reference matrix, showing symbols explicitly referenced by module (RXVP, QLAB)
- Symbol access matrix, showing accessibility of each symbol to modules and indicating those symbols which are (1) explicitly referenced and (2) implicitly accessed through actual invocation parameter lists (RXVP, QLAB)
- Symbol control structure reference matrix, showing symbols explicitly referenced in control statements (e.g., IF, GOTO, SWITCH)

## <u>Control Structure Analysis</u>

The identification of the control structure embodied in software is necessary to other AVS functions. Structural analysis determines the possible program control flow by identifying the sequential code segments which define decision-to-decision paths (DD-paths) with each module and by locating all inter-module invocations and control transfers. This information is the basis of static analysis of intermodule structures, testing coverage analysis and assistance, data flow analysis, and other analyses which require knowledge about program flow. As part of its function, the structural analysis processor (currently part of JAVS, RXVP and QLAB) includes checks on the control structure of the software such as identification of:

- Structurally non-terminating modules
- Structurally unreachable code
- Direct transfers into a deeper or alternative control nesting level ("borrowed code")
- Direct out-of-module transfers within the module invocation hierarchy ("escape" path)
- Direct out-of-module transfers to another chain in the module invocation hierarchy ("threaded" path)
- Circular module invocation (recursion)

These constructs, although permitted in the language, may not (or cannot) be detected by normal compiler operation. Other structural services currently provided in existing AVS implementations include:

- Reaching set and reaching sequence analysis for a specified target statement or path (JAVS, RXVP, QLAB)
- Program restructurizing to a standard form (i.e., single-entry, single-exit structured modules) (RXVP, QLAB)

## Consistency Checking

Although most languages require software to be internally consistent, many compilers do not (or cannot) check for consistency. Currently, an AVS with access to the entire software source can perform valuable checks for consistency such as:

- Agreement between actual and formal parameter lists for number of arguments and argument specifications (RXVP, QLAB)
- Consistent use of syntactic types (e.g., no implied type conversion, legal operations, proper number of subscripts) (RXVP, QLAB)
- Consistency in redundant declarations (e.g., matching data structure definitions in independent compilation units)
- Reasonable variable usage (e.g., a variable must be set before use and used after set; all iterations should modify escape control variables) (RXVP, QLAB)

With the addition of semantic information in the software beyond what is needed for execution, consistency checking can be extended to include:

- Checking of physical properties of data with its usage (e.g., dimensions: length, time, temperature; units: feet, milliseconds, degrees Celsius; and range: $0 \leq x \leq 100$ feet) (QLAB)
- Actual versus asserted variable usage (e.g., a module's input variables are used and its output variables are set) (QLAB)
- Violation of intended data access rights, permitted operations, and permanence.
- Verification of the sufficiency of a formal specification of a module's function to support a correctness proof.

All of these require a mechanism for formally stating the additional information (e.g., a language extension); most use the additional information in conjunction with normal source code.

## Program Proof of Correctness

Logical assertions are specifications on the values of data. These assertions can be used to show that a program is consistent with its specification. They are the basis for dynamic checks on program performance and for formal proofs of correctness. A verification condition is the combination of the logical assertions and the program operations on data for a particular program path. In order to verify a program using formal logic it is necessary to generate verification conditions and show that they are true.

Assertions are made for a module's entry and exit points and at intermediate locations (e.g., as loop invariants, invocations of lower level modules). Verification conditions can be generated all together for a complete module or separately at structural boundaries. Using path analysis techniques and symbolic execution, verification conditions are automatically generated by QLAB.

There are two automatable techniques for determining the truth or falseness of a verification condition: one is to use an automatic theorem prover and the other is to use a set of simplification rules to reduce the expression to true or false. Automatic simplification has been accomplished by theorem provers, but has required large quantities of computer time and memory. Furthermore, automatic provers have been applied primarily to very small programs with restricted language subsets (e.g., real integers, addition and multiplication operations, simple loops and conditional control structures). It is generally accepted that simplification by the application of a set of rules is the best approach to take. Interactive verification condition generation and simplification is being explored in QLAB with application to real programs, and results are encouraging.

### 5.1.2 Dynamic Tools

Dynamic AVS tools support analysis of execution characteristics of a program (i.e., the dynamic properties of program behavior).

Current AVS implementations have emphasized the testing of software to achieve higher quality. There is a wide variety of testing analysis tools which can be used on existing software such as:

- Control structure instrumentation which records the dynamic flow of program control through insertion of probes (JAVS, RXVP)
- Coverage analysis which reports the program paths exercised (or not exercised) during testing in varying levels of detail (JAVS, RXVP)
- Performance analysis which reports timing of modules (JAVS)
- Executable assertions about program behavior manually inserted in existing software producing instrumentation probes which report violations of the assertions during test execution (JAVS, RXVP, QLAB).

## 5.2 FUTURE AVS CAPABILITIES

Compared to today's AVS, tomorrow's AVS will be a far more effective tool for the software developer. It will offer expanded capabilities and will be both easy to use and cost-effective. The developer will design and implement software to take advantage of the AVS in much the same way that current software is designed and implemented to take advantage of current tools--higher level language compilers, system loaders, libraries, text maintenance tools and operating system processors. Existing languages will be extended and new languages developed to improve software quality. Some of these language features will supply additional information for analysis by the AVS.

In the following subsections, capabilities recommended for advanced AVS tools are described. These are described in terms of static and dynamic capability.

### 5.2.1 Static Tools

The basic emphasis in static checking for advanced systems should be directed toward static analyses that check for erroneous usage compared to declared usage. The declaration may be supplied by the user through expanded source language (e.g., more restrictive variable typing), or the declared usage may be implicit in a set of coding standards. The categories discussed are:

- Syntactic documentation
- Coding standards checking
- Data flow analysis

The recent experiences with AVS applications have indicated other useful syntactic documentations attainable with existing technology:

- Explicit and implicit module invocation matrix, showing both direct and (possible) indirect invocation of each module to other modules, whether or not known to the AVS
- Scope data definition structure, showing scope hierarchy data definition (e.g., as determined by module nesting structure)
- Symbol control structure reference matrix, showing symbols explicitly referenced in control statements (e.g., IF, GOTO, SWITCH)
- File symbols reference matrix, showing I/O interfaces
- Symbol file data transfers, showing symbols referenced in I/O data transfers

Although none of the above actually requires extensions to the language, some mechanism to designate a module's membership in one or more selected sets would simplify AVS usage. (Current AVS tools such as JAVS provide for selecting a list of modules by name from the known modules under user command.)

## Coding Standards Checking

The enforcement of coding standards is largely a preventive measure used to avoid problems often associated with poor language usage. Standards checkers analyze code for conformance and identify offending statements. Standards usually are imposed on module size (number of executable statements), permitted language constructs (language subset), comments (all control statements preceded by descriptive commentary), and symbol naming conventions (all modules have long names, mnemonic data symbols, increasing sequential order for statement labels). A common technique for enforcing standards is to use a standards checker as a preprocessor to the compiler and prevent nonconforming modules from being compiled. With an AVS, however, standards checking can be integrated along with other static analysis tools, such as data flow analysis described below.

## Data Flow Analysis

Improper use of program variables is a major source of errors in large software systems. Data flow analysis classifies the usage of all program variables as input (to supply a value) or output (to receive a value). The analysis checks variable usage in all statements including initialization, decision, read and write statements. Parallel analysis of all variables in a module can be performed, starting with the modules at the bottom of the system calling tree and working to the top of the tree. The control structure of the module is used to perform the analysis.

Each module and each statement in a module needs to be analyzed only once. After a module has been analyzed, it can be represented in terms of its variables and their usage rather than as a set of statements with an inherent structure.

Loop termination problems can also be located by the use of data flow analysis techniques. Paths can be identified in loops which do not alter the loop control variable. Such paths are the potential causes of infinite loops.

Data access statements (INPUT, OUTPUT) qualify or limit the access rights and operations on data by explicitly specifying the input and output variables. These statements can be added to the source program and processed by the AVS to provide expected and actual usage of variables. These two data access statements, together with the source code itself, have a clear correspondence to the "stimulus-action-response" model of software functions which is being examined as a means for representing requirements.

## 5.2.2 Dynamic Tools

In thinking toward the future, there are two general categories of dynamic tools: those which are concerned with testing software and those which support software fault tolerance.

### Testing

Advanced AVS implementations will offer not only extensions of analyses currently developed (Sec. 5.1.2) but also tools which support test environment generation and test history maintenance. Dynamic consistency checking is one type of expanded analysis capability which can be used to expose errors not found with static consistency checks. These dynamic checks include detection of violations of:

- Subscript limitations
- Declared variable ranges
- Timing constraints
- Set before use and no use after set practices

An important capability for advanced AVS implementation is automated test environment generation. With current AVS capabilities the tester must construct the test environment manually. He assembles the code to be tested from instrumented and uninstrumented modules, sometimes producing new software for test drivers and module stubs to make a complete program. Test data is also manually prepared. He may use reports from the AVS to identify what modules are necessary, what interfaces must be supplied, and what variables are directly affected by test data inputs. In this respect the AVS is a passive participant in the testing activity.

A more active role for the AVS in automated test driver and stub generation and test data generation is certainly desirable. For example, data flow analysis coupled with use of module input and output declarations and entry and exit assertions can provide the basis for automated test environment support.

For a target module, a test driver can be constructed which contains a synthesis of the module's communication space, module invocations, and code to initialize input data; stubs for other modules invoked by the target module can be similarly constructed. The tester then supplies the required data, entry assertions verify the data; exit assertions verify the test results.

Systematic software testing often requires keeping careful records of the testing history of software systems: what tests were done, what test data was used, what was accomplished. This information is used as a test management device to select test strategies which minimize the amount of additional testing. If the software is modified, previous test records can be used to select test data and software configurations for retesting activities. Current AVS implementations permit the tester to identify each test, but the AVS makes no direct use of the information. Record-keeping is largely a clerical process that, if manually done, is prone to error. Although automated test history maintenance can involve retention of large quantities of information (e.g., detailed coverage data), some automatic mechanism for identifying, extracting and retaining essential data is desirable for advanced AVS implementation.

### Software Fault Tolerance

Another approach to improving software quality is to provide a mechanism whereby software can continue to function properly although some failure (hardware, software, or bad data) has occurred. Fault tolerance attempts to increase the reliability of a system through dynamic redundancy. A fault-tolerant software system requires redundant software or hardware in order to

1. Detect faults before major damage occurs
2. Diagnose faults so repair may be performed
3. Recover from faults by re-establishing an acceptable system state

Correctness proofs assume that no hardware faults will occur and that only specified data inputs will be processed. For this reason "correct" systems may not be reliable systems. Fault-tolerant techniques assume that hardware faults will occur, data input specifications will be violated, and software errors are inevitable. A variety of approaches for attaining fault tolerance in software are currently being investigated. All use various combinations of software and/or hardware for support. This has strong implications for the role of AVS in testing: some mechanism must be provided to simulate faults in order to test error detection, repair, and recovery functions.

### 5.2.3 New AVS Requirements

There are a number of ongoing developments in the computer arena which impact the requirements for AVS capabilities. Among these are

- New software languages and software designs (e.g., to support concurrency, fault tolerance, restricted data access)

- Non-conventional hardware architectures (e.g., tagged architecture, distributed data processing)

- System software which supports new developments in hardware and application software

All of these factors will impose new requirements on AVS facilities.

Current AVS implementations have been designed for a stable environment of rather conventional characteristics: e.g., a large host computer, with ample auxiliary storage for a substantial data base, and a computer operating system supporting large program applications. This has been an adequate assumption for experimental AVS usage. The need for AVS adaptation to more restricted host environments (e.g., to mini-computers) has now become evident. This impacts the design of AVS software itself. For example, in more restrictive environments, each AVS tool could be separately implemented for the most efficient use of computer resources.

Many AVS functions closely parallel those of current compilers: for example, syntactic source analysis. An AVS developed independently of a compiler does offer an independent check on these functions. This, however, is also a drawback since erroneous conflicts in interpretations of source text between the AVS and the compiler are a source of difficulty in AVS usage. It is not at all unreasonable to expect advanced AVS implementations to make use of compiler source text analysis functions in constructing portions of the data base. This can be accomplished by requiring the compiler to retain the information in an acceptable form for input to an AVS.

## APPENDIX A

## RADC TEST PLAN FOR JAVS IMPLEMENTATION CONTRACT

## A.1 INTRODUCTION

The purpose of the current "JAVS Implementation" effort is to perform an evaluation of the JOVIAL Automated Verification System (JAVS) by using it to test and verify a large, operational computer software system.

The goals of this effort are to:

a. Identify both strong and weak areas of the JAVS when used to process large software systems.

b. Verify that all portions of the JAVS are operating properly and that current documentation is accurate. Deficiencies found in the JAVS during the testing process will be corrected if the resources required are within the limits of the effort.

c. Identify and recommend areas of the JAVS which could be subject to future enhancement.

d. Document the results of the testing process for future reference and study.

e. Establish recommended procedures for using the JAVS System to test and verify large software systems.

The COMPOSE and QUERY components of SAC's Force Management Information System (FMIS) were initially chosen as candidates for the test because they met size, complexity, language type and availability requirements.

As a secondary benefit of this effort, portions of the FMIS software will be tested and documented. Emphasis will be placed on testing and verifying the COMPOSE subsystem for the following reasons:

a. COMPOSE is much larger and more complex than QUERY

b. COMPOSE incorporates 50% of the modules in the QUERY subsystem

c. QUERY software has recently been reviewed manually by SAC personnel. Much of the erroneous code found has been removed, resulting in a more efficient operation.

During the testing process, it will be extremely valuable to identify erroneous code and/or coding techniques and make recommendations for pointed evaluations or enhancements of the COMPOSE software.

## A.2 JAVS TEST REQUIREMENTS

Using FMIS as a test program, perform the following:

a. STEP1:

Demonstrate language coverage. Document any limitations found in both the software and hardware resources. Document processing time and resources (e.g., CP and PP time, lines of source code, library size, lines of and number of modules). Document any side-effects of STEP1 (e.g., Alter Library). Analyze feasibility of a rewritten front-end. Make recommendations.

b. STEP2:

Analyze behavior of DD-path generation. Document limitations resources used, side effects, etc. Analyze feasibility of expanding STEP2 constraints.

c. STEP3:

Demonstrate instrumentation capabilities. Use and analyze directives as testing aids and debugging aids. Document resources required (especially monitors use in JAVS directives), limitations, and side effects. Consider potentialities of directives in documentation and as "descriptive" execution.

d. STEP4:

Analyze usefulness of reaching sets. Consider alternatives or enhancements to reaching sets and control flow picture. Document limitations, resources, and side-effects.

e. STEP5:

Consider enhancements to reports. Consider provision of JAVSTEXT selection for library reports. Document results.

f. Test Execution:

Determine if current limitations are practical. Analyze resources to determine if probe routines should be simpler and save summarizing test case totals for STEP6. Document loading sequence (JCL) for overlay and non-overlay user programs.

g. STEP6:

Demonstrate limitations of coverage reports (e.g., in unstructured inter-module action, etc.). Document usefulness and inability of coverage reports to aid in retesting. Document resources, etc.

h. JAVS Overlay Version:

Demonstrate capability to perform all processing steps. Indicate possible improvements in overlay configuration. Analyze size/speed tradeoffs. Document side effects, etc.

i. Macro Processor:

Consider enhancements. Demonstrate performance (proper command expansion). Demonstrate and document all feasible combinations of macros and standard commands.

## A.3 JAVS-FMIS INTEGRATION

The following steps are necessary to incorporate the JAVS execution coverage capability into the COMPOSE software (see Fig. A.1).

1. Determine naming conventions for instrumented source files, object files, compilation JCL, execution JCL, audit files and libraries.

2. Modify existing compilation JCL: for each start-term sequence to be tested, the compilation JCL must be modified such that it compiles the instrumented source text and outputs the object on specially designated file. In addition, the JAVS probe COMPOOL must be included.

3. Modify existing execution JCL: for each start-term sequence to be tested, the execution JCL must be modified such that it executes the object file as output from the compilation of the instrumented source text. In addition, the selection of the JAVS probe software must be included.

Upon completion of the integration, the following files should exist for each start-term sequence of COMPOSE that is to be tested:

- SAC symbolic file
- SAC binary file
- SAC compilation JCL
- instrumented symbolic file
- instrumented object file
- modified compilation JCL
- modified execution JCL

## A.4 TESTING PROCESS

### A.4.1 <u>Library Creation</u>

The initial step in the testing process will be to create the library or libraries required to contain all of the COMPOSE software to be tested. This operation is performed by execution of the JAVS/STEP1. Following STEP1, STEP2

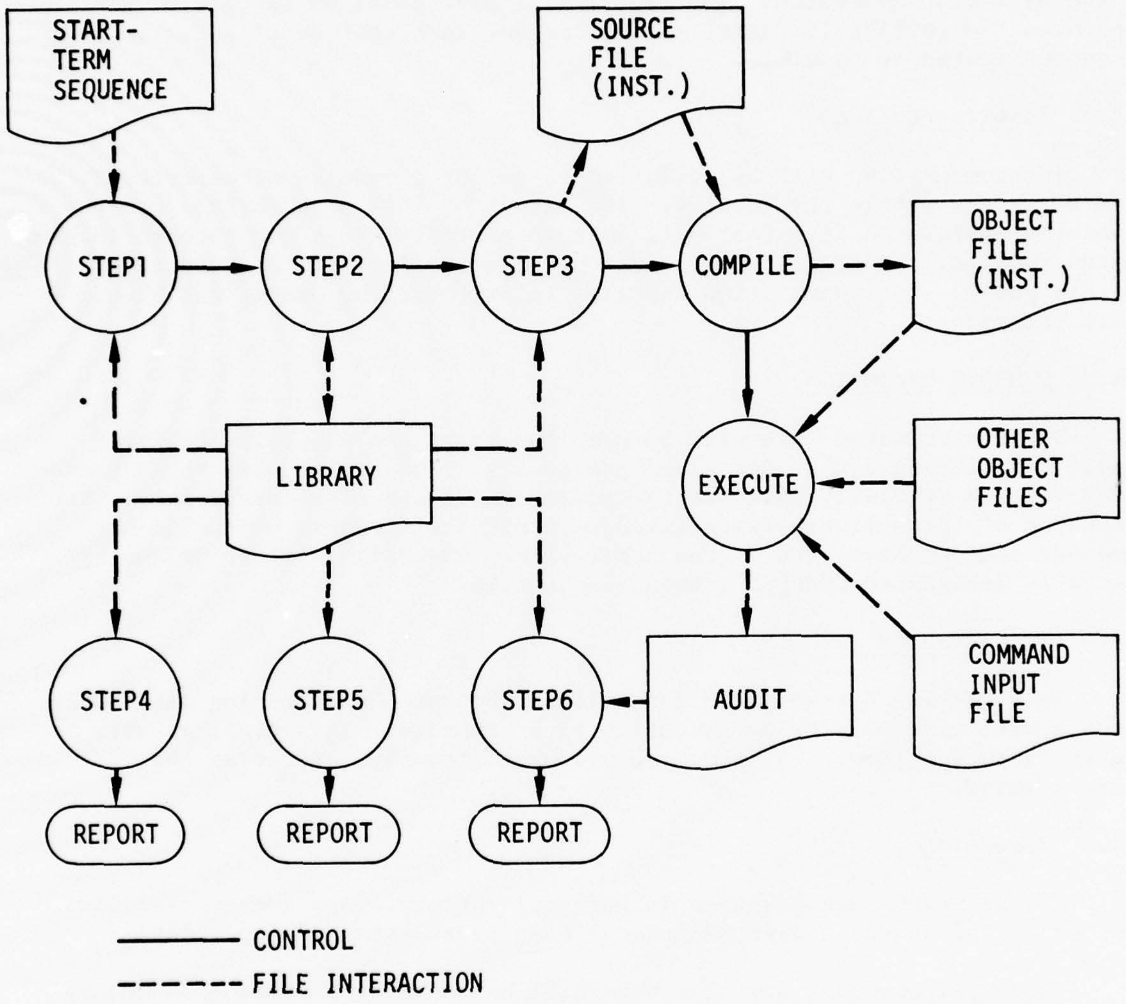


Figure A.1. JAVS Processing Steps

will be performed to provide the structural analysis. STEP1 and STEP2 may be performed together by use of the the BUILD LIBRARY macro command.

If possible, one large library containing all START-TERM sequences to be tested will be created. The advantage of one large library versus many smaller ones is that the STEP5 reports (intermodule dependency) will cover all of the system to be tested. The report will give clues as to what START-TERM sequences the smaller libraries should include such that testing can be kept as uncomplicated as possible.

## A.4.2 Instrumentation

Instrumentation will be performed on selected modules of selected START-TERM sequences within the library. The output of this step (STEP3) will be a specially designated file that will contain a START-TERM sequence with instrumented modules. PROBI statements must be inserted into the instrumented file at this point. Instrumentation may also be performed by use of the PROBE macro command.

## A.4.3 COMPOSE Execution

The instrumented file will be compiled using the specially tailored module compilation JCL as described previously. The system will then be executed via the specially tailored execution JCL as described previously. The execution of the software (with imbedded probes) will cause execution path coverage data to be stored in the AUDIT file. Execution will be driven by a specially designated COMPOSE command input file.

## A.4.4 Test Results

Test results on the execution will be obtained by executing JAVS/STEP6. Reports will give results on execution path coverage. This step, as well as system execution (Sec. A.4.3) may be performed together by use of the TEST macro command.

## A.4.5 Retesting

If execution path coverage is not satisfactory, then system execution must be performed again with additional test cases (command input data).

The development of new test data will be aided by reports produced by JAVS/STEP4 (module test assistance and segment analysis) and JAVS/STEP5 (retest guidance and analysis). STEP4 and STEP5 may be performed together by use of the DOCUMENT macro command.

Complete testing and verification of COMPOSE will require Secs. A.4.2, A.4.3, and A.4.4 above to be performed for each START-TERM sequence on the library.

# APPENDIX B

# ERROR ANALYSIS

## B.1 JAVS ERRORS UNDETECTED DURING SELF-TEST

1. Instrumented code expansion. This program logic error occurred when instrumenting modules for invocations and returns only. During the code expansion of a switch statement containing a label, the length of the buffer containing the probe line was not adjusted for the label.

2. DD-path summary value. This coding error was the incorrect variable name used in accumulating the summary of DD-paths during Test Execution. The error manifested itself during self-testing but was not noticed in the output.

3. Infinite loop in module selection. This program logic error resulted when the user specified an incorrect JAVSTEXT name for a given module name.

## B.2 DESIGN ERRORS

1. Labels on BEGIN statements. JOCIT and J3 JOVIAL allow statements labels on BEGIN statements. The syntax analyzer did not recognize this construct and was modified to accept it.

2. Comment processing. The syntax analyzer would produce erroneous code when JOVIAL comments contained certain keywords and ideograms. The syntax analyzer has been corrected to ignore any of these constructs, unless the comment is a JAVS directive.

3. Overlay variables. If the user was not processing the COMPOOL containing the declarations for variables used in overlay statements, the syntax analyzer would produce unnecessary error messages and cause the first executable statement number to be set incorrectly. The syntax analyzer was modified so it does not check the overlay COMPOOL-defined variables.

4. Direct code. Indented to the first indentation level on punch file (used for compilation).

## B.3 BOUNDARY CONDITIONS ERRORS

1. Array overflow. This occurred in producing a JAVS cross reference on more than 83 modules. Because of the common block's load sequence, the error was detected only in the overlay version.

2. Searching beyond table limit. This error occurred in two locations of the code and was corrected by checking the length of the given table before searching.

The first release of JAVS documented the constraint of not allowing transfers to other modules via a GOTO or SWITCH statement. This constraint was based on three facts: it is a poor programming practice, the syntax analyzer does not provide the necessary information to determine the defining module of the label, and the concept of the DD-path does not permit transfers into the middle of a DD-path (as is likely via an external label). Nevertheless, this design change was requested and made. Testing the modified code uncovered five errors due to the code changes.

## B.4 COMPILER ERRORS

1. Comparisons. Version 042275 of the JOCIT compiler compares only the first six characters in strings, regardless whether the strings are declared to be longer or if byte operations are used.

2. Character functions. The compiler does not save the temporary value of functions which return more than a single machine word character string. JAVS source code was modified to avoid this error in one instance, but the source code should be recompiled using the newer version of the compiler as soon as it is released.

## B.5 JAVS CONSTRAINTS

1. Comments imbedded in switches. The Reference Manual now includes the constraint that BASIC, COMMENTS = OFF be specified when switch declarations contain imbedded comments.

2. External TEST variable. A constraint was added to the Reference Manual specifying that a TEST variable cannot be used outside the module containing the FOR loop statement to which it refers.

3. BEGIN-END requirement. External CLOSE modules require a BEGIN-END construct surrounding the executable text. The compiler already requires the construct in all other module forms.

Each of the above constraints produce JAVS messages when violated.

4. The external CLOSE statement must be removed to build a JAVS library and returned for compilation.

## B.6 OTHER CAUTIONS

1. One of the COMPOSE segments approached the JOCIT compiler limit of 1,000 COMPOOL and externally defined variables. Additions of the four data collection routine names pushed the total number over the limit.

2. Instrumentation converts a JOVIAL IF statement to an IFEITH. These extra characters caused the continuation line to start with a $ in several instances of COMPOSE testing. GCOS treats any input line starting with a $ to be a control card. A compilation error will result. The tester can either move the character (along with a preceding left parenthesis in the case of a left subscript bracket) or use the GCOS control cards:

```
$       DATA    S*,,COPY,ENDFC
$       SELECTD (file containing instrumented code)
$       ENDCOPY S*
```

during compilation of the instrumented source code.

## APPENDIX C

## JAVS COMMAND SUMMARY

| JAVS COMMANDS (DEFAULTS UNDERLINED) | STEP |
|---|---|
| ALTER LIBRARY = <libname>. | (Universal) |
| ANALYZER. | ANALYZER |
| ANALYZER,ALL. | ANALYZER |
| ANALYZER,ALL MODULES. | ANALYZER |
| ANALYZER,CASES = <number>. | ANALYZER |
| ANALYZER,DDPATHS. | ANALYZER |
| ANALYZER,DDPTRACE. | ANALYZER |
| ANALYZER,FACTOR = <percent-increase>. | ANALYZER |
| ANALYZER,HIT. | ANALYZER |
| ANALYZER,MODLST. | ANALYZER |
| ANALYZER,MODTRACE. | ANALYZER |
| ANALYZER,MODULE = <name-1>,<name-2>,...,<name-n>. | ANALYZER |
| ANALYZER,NOTHIT. | ANALYZER |
| ANALYZER,SUMMARY. | ANALYZER |
| ANALYZER,TIME. | ANALYZER |
| ASSIST,CROSSREF,JAVSTEXT = <text-name-1>,<text-name-2>,..., <text-name-n>. | ASSIST |
| ASSIST,CROSSREF,LIBRARY. | ASSIST |
| ASSIST,PICTURE. | ASSIST |
| ASSIST,PICTURE{,CONTROL}{,NOSWITCH}. | ASSIST |
| ASSIST,REACHING SET,<number-to>{,<number-from>} {,PICTURE{,ITERATIVE}}. | ASSIST |
| ASSIST,STATEMENTS. | ASSIST |
| BASIC. | BASIC |
| BASIC,CARD IMAGES = ON/OFF. | BASIC |
| BASIC,COMMENTS = ON/OFF. | BASIC |
| BASIC,DEFINES = ON/OFF. | BASIC |
| BASIC,ERRORS = ON/OFF/LIMIT/TRACE. | BASIC |
| BASIC,SYMBOLS = ON/OFF/PARTIAL. | BASIC |
| BASIC,TEXT = COMPUTE/BOTH/PRESET/JAVSTEXT. | BASIC |
| *BUILD LIBRARY {= <library name>}, | BASIC, STRUCTURAL |
| CREATE LIBRARY = <libname>. | (Universal) |

*Can be used only with the overlay version.

| JAVS COMMANDS (DEFAULTS UNDERLINED) | STEP |
|---|---|
| DEPENDENCE,BANDS. | DEPENDENCE |
| DEPENDENCE,BANDS = <number>. | DEPENDENCE |
| DEPENDENCE,GROUP,AUXLIB. | DEPENDENCE |
| DEPENDENCE,GROUP,LIBRARY. | DEPENDENCE |
| DEPENDENCE,GROUP,MODULES = <name-1>,<name-2>,...,<name-n>. | DEPENDENCE |
| DEPENDENCE,PRINT,INVOKES. | DEPENDENCE |
| DEPENDENCE,SUMMARY. | DEPENDENCE |
| DEPENDENCE,TREE. | DEPENDENCE |
| DESCRIBE = ON/OFF. | (Universal) |
| *DOCUMENT{,JAVSTEXT=<text-name>{,MODULE=<name-1>,...}}. | ASSIST, DEPENDENCE, (Universal) |
| END. | (Universal) |
| END FOR. | (Universal) |
| FOR JAVSTEXT. | (Universal) |
| FOR LIBRARY. | (Universal) |
| FOR MODULE = <name-1>,<name-2>,...,<name-n>. | (Universal) |
| INSTRUMENT. | INSTRUMENT |
| INSTRUMENT,MODE = INVOCATION/DDPATHS/DIRECTIVES/FULL. | INSTRUMENT |
| INSTRUMENT,PROBE,DDPATH = <probe-name>. | INSTRUMENT |
| INSTRUMENT,PROBE,MODULE = <invocation-name>. | INSTRUMENT |
| INSTRUMENT,PROBE,TEST = <test-name>. | INSTRUMENT |
| INSTRUMENT,STARTTEST = <modname>,<textname>,<stmt. no.> {,<TESNAM>}{,<TFLAG>}. | INSTRUMENT |
| INSTRUMENT,STOPTEST = <modname>,<textname>, <stmt. no.>. | INSTRUMENT |
| JAVSTEXT = <text-name>. | (Universal) |
| MERGE. | (Universal) |
| MODULE = <name>. | (Universal) |
| OLD LIBRARY = <libname>. | (Universal) |

*Can be used only with the overlay version.

| JAVS COMMANDS (DEFAULTS UNDERLINED) | STEP |
|---|---|
| PRINT,DDP. | (Universal) |
| PRINT,DDPATHS. | (Universal) |
| PRINT,DMT. | (Universal) |
| PRINT,JAVSTEXT = <text-name-1>, INSTRUMENTED = ALL. | (Universal) |
| PRINT,JAVSTEXT = <text-name>,INSTRUMENTED = <name-1>, <name-2>,...,<name-n>. | (Universal) |
| PRINT,JAVSTEXT = <text-name>. | (Universal) |
| PRINT,MODULE. | (Universal) |
| PRINT,SB. | (Universal) |
| PRINT,SDB. | (Universal) |
| PRINT,SLT. | (Universal) |
| PRINT,STB. | (Universal) |
| *PROBE{,JAVSTEXT = <text-name>{,MODULE = <name-1>,...}}. | INSTRUMENT, (Universal) |
| *PROBI,STARTTEST = <modname>,<textname>,<stmt. no.> {,TESNAM}{,TFLAG}. | INSTRUMENT, (Universal) |
| *PROBI,STOPTEST = <modname>,<textname>,<stmt. no.>. | INSTRUMENT, (Universal) |
| PUNCH,JAVSTEXT = <text-name>. | (Universal) |
| PUNCH,JAVSTEXT = <text-name>,INSTRUMENTED = ALL. | (Universal) |
| PUNCH,JAVSTEXT = <text-name>,INSTRUMENTED = <name-1>, <name-2>,...,<name-n>. | (Universal) |
| PUNCH,MODULE. | (Universal) |
| START. | (Startup) |
| STRUCTURAL. | STRUCTURAL |
| STRUCTURAL,PRINT = SUMMARY/DEBUG. | STRUCTURAL |
| *TEST{,MODULE = <name-1>,<name-2>,...<name-n>}. | ANALYZER, (Universal) |

*Can be used only with the overlay version.

## APPENDIX D

## JAVS ENHANCEMENTS

Enhancements to JAVS were made primarily to ease the burden of the user and to extend certain limits found during the COMPOSE testing. These enhancements are as follows:

1. Provide automatic insertion of the test initiation and termination invocations. These calls to the PROBI data collection routine can be specified by the user through the commands:

   ```
   INSTRUMENT, STARTTEST = ....

   INSTRUMENT,STOPTEST = ....
   ```

2. The PROBI data collection routine issues a message to the AUDIT file (printed during post-test analysis) indicating the test termination has occurred. Data collection arrays and counters are cleared and subsequent execution results are saved, if processing continues. At this time, post-test analysis processes only the first set of testcases (one test run).

3. The post-test analysis SUMMARY report was augmented to include in a separate box, all modules not invoked during the test, but which the user had specified. The DD-path coverage percentage is recomputed and printed to reflect the additional modules.

4. A macro command processor was developed to allow the user to document and test his software using only four macro commands: BUILD LIBRARY, PROBE, DOCUMENT and TEST. The macro commands can be used only with the overlay version of JAVS. Macro and standard commands can be intermixed in the same command set.

5. The module selection commands FOR ALL and FOR ALL MODULES were changed to: FOR LIBRARY and FOR JAVSTEXT, respectively. The command processor recognizes both forms of the selection commands, but only the latter form appears in the revised documentation.

6. The processing step declaration command (e.g., STEP = 1.) has been removed from the standard version of JAVS, making it more consistent with the overlay version.

7. Summary values for the number of probe lines, program statements, executable statements, and DD-paths for the library are calculated and printed in the JAVS wrapup report.

8. The DD-path control flow picture was modified to print the DD-path numbers in order of their left-to-right appearance in the diagram. Whe too many paths are computed to be printed, the message "missing DD-paths," was changed to "paper exceeded." This message appears on the side of the picture where the overflow occurs.

9. The number of modules selected in FOR LIBRARY or FOR JAVSTEXT was increased from 100 to 150.

10. The number of allowable DEFINE statements in a module was increased to 250.

11. A warning message is produced whenever transfer occurs to a label not defined in the module (external label). The message includes the label's name and the statement number and module name of the transfer statement.

## APPENDIX E

## GLOSSARY OF AVS TERMINOLOGY

AVS. Automated Verification System.

AVS Database. In an AVS, the collection of information maintained internal to to the AVS and which contains all pertinent data about all modules known to the AVS.

Actual Parameter. In an invocation of a module, the set of variable names passed to the invoked module in the actual parameter list.

Automated Verification System. A system for the analysis of software systems oriented toward systematic, comprehensive testing (exercise) as a means to perform software verification. JAVS is an example of such a system.

Bottom-up Testing Strategy. A systematic testing philosophy which seeks to test those modules at the bottom of the invocation structure earliest.

Collateral Testing. Collateral testing is that testing coverage which is achieved indirectly, rather than as the direct object of a testcase activity.

Communication Space. The communication space of a module consists of those symbols, known within the module, by which information can be passed to or from the module. Communication space mechanisms consist of formal parameters, global variables, and return parameters.

Computation Directive Instrumentation. The process of producing an altered version of a module as the result of user-inserted directives which is logically equivalent to the unmodified module, but which contains executable code that provides for collecting information about the dynamic behavior of the module during its execution.

Computation Structure. The operation of the program on the data.

Control Level. (Control nesting level.) The control structure of a program is hierarchical. At the module's entry, the control level is 0. Each new DD-path or beginning of a BEGIN-END block increases the control level; the end of each DD-path or BEGIN-END block decreases the control level.

Control Structure. The set of program statements which alter the normal sequential flow from one statement to the next.

Data Structure. Organization of the data on which the program operates.

DD-Path. A DD-path, or decision-to-decision path, is the set of statements in a module which are executed as the result of the evaluation of some predicate (conditional) within the module. The DD-path should be thought of as including the sensing of the outcome of a conditional operation and the subsequent executions up to, and including, the computation of the next predicate value but not including its evaluation.

DD-Path Instrumentation. The process of producing an altered version of a module which is logically equivalent to the unmodified module but which contains calls to a special data collection subroutine which accepts information as to the specific DD-path sequence incurred in an invocation of the module.

DD-Path Predicate. A logical formula involving variables/constants known to a module and, possibly, the values .TRUE. and .FALSE., which must be satisfied for the DD-path to be executed.

Decision Statement. A decision statement in a module is one in which an evaluation of some predicate is made which (potentially) affects the subsequent execution behavior of the module.

Decision-to-Decision Path. See DD-path.

Directive. A user-supplied statement imbedded in the source text which directs the AVS to perform a specified function.

Essential DD-path. A DD-path in a reaching set which must be executed in order to reach the designated DD-path.

Executable Statement. A statement in a module which is executable in the sense that it produces object code instructions.

External Label. See global label.

Flow. A particular sequence of DD-paths.

Formal Parameter. For an invokable element of program text, the set of variable names which are assigned value outside of the program text.

Functional Specifications. A set of behavior and performance requirements which, in aggregate, determine the functional properties of a software system.

Functional Test Cases. A set of test case datasets for software which are derived for testing specific tasks or functions.

Global Label. A statement label residing in one module but which is transferred to from other modules.

Global Variable. In a module, a global variable is one which may receive a value as the result of actions outside the module.

Input Domain. See input space.

Input Space. The input space of a module consists of that subset of a module's communication space which can be (1) altered externally to the module, and (2) which is (potentially) used within the module in a way that affects its execution.

Instrumentation. The automatic insertion of software probes (e.g., invocations to data collection routines) to capture information during execution.

Intermodule Dependencies. Generally refers to module interaction via invocations but can also include interaction due to external label transfers.

Invocation Point. The invocation point of a module is the first statement in the module (in JOVIAL, a program, procedure, or close), or, if the module has multiple entry points, an entry statement.

Invocation Structure. The hierachy of invocations of one module by another within a software system.

Iterative Flow. Iterative flow is represented by a sequence of DD-paths with the property that some DD-path belonging to the sequence can be executed one or more times.

JOCIT JOVIAL. (JOVIAL Compiler Implementation Tool). A dialect of JOVIAL/J3.

JOVIAL. Unless further specified, any of the dialects of the family of JOVIAL languages.

JOVIAL/J3. The JOVIAL programming language, J3 subset, as defined in Air Force Manual AFM-100-24.

Memory. A module is said to have memory if there is some interior code condition which makes it possible to execute some DD-path only by making two or more invocations of the module.

Memory Space. The memory space for a module consists of those cells known to the module which allow it to have memory (see Memory).

Module. A module is a separately invokable element of a software system.

Non-Executable Statement. A declaration or directive within a module which does not produce (during compilation) object code instructions directly.

Output Space. The output space of a module consists of the collection of variables, including file actions, which are (or could be) modified by some invocation of the module.

Path. A sequence of DD-paths.

Predicate. A formula involving variables/constants and relational operators, which can be evaluated to .TRUE. or .FALSE..

Program Validation. The process of developing and verifying the correspondence between an implemented software system and the set of functional specifications which correspond to it.

Program Verification. The process of verifying that a set of functional test cases meets structural testing goals.

Reaching Set. The set of all DD-paths that connect together to form paths from one designated DD-path to another.

Software System. A collection of modules, possibly organized into components and subsystems, which solves some problem.

Software Validation. See Program Validation.

Structural Instrumentation. Instrumentation of module invocation entries, exits and DD-paths without changing the logic of the uninstrumented source code.

Structural Test Cases. A set of test case patterns, derived from the control structure of a module (or a collection of modules). The combination of a structural test case and appropriate program input data results in a functional test case.

Test. A test is one or more unit tests of one or more modules.

Test Case. See Test.

Test Case Dataset. A test case dataset is a specific set of values for variables in the *communication space* of a module which are used in a test.

Testing Coverage Measure. A measure of the testing coverage achieved as the result of one unit test, usually expressed as a percentage of the number of DD-paths within a module which were traversed in the test.

Test Execution. Execution of the test object in which one or more modules of the test object have been instrumented. Output differs from normal execution output in that information captured from the instrumented code is written to a sequential file for later analysis.

Test Object. The software to be tested.

Testing Stub. A testing stub is a module which simulates the operations of a module which is invoked within a test. The testing stub can replace the real module for testing purposes.

Testing Target. The current module (system testing) or the current DD-path (unit testing) upon which testing effort is focused.

Testing Verification. See program verification.

Top-down Testing Strategy. A systematic testing philosophy which seeks to test those modules at the top of the invocation structure earliest.

Unit Test. A unit test of a single module consists of (1) a collection of settings for the input space of the module, and (2) exactly one invocation of the module. A unit test may or may not include the effect of other modules which are invoked by the module undergoing testing.

Unreachability. A statement (or DD-path) is unreachable if there is no logically obtainable set of input-space settings which can cause the statement (or DD-path) to be traversed.

## REFERENCES

1. E. F. Miller, Jr., Methodology for Comprehensive Software Testing, General Research Corporation CR-1-465, June 1975.

2. C. Gannon, N. B. Brooks, and R. J. Urban, JAVS Technical Report, Vol. 1, "User's Guide," General Research Corporation CR-1-722, RADC-TR-77-126 dated April 1977.

3. C. Gannon, N. B. Brooks, JAVS Technical Report Vol. 2, "Reference Manual," General Research Corporation CR-1-722, RADC-TR-77-126 dated April 1977.

4. N. B. Brooks and C. Gannon, JAVS Technical Report, Vol. 3, "Methodology Report," General Research Corporation CR-1-722, RADC-TR-77-126 dated April 1977.

5. Contract Data Requirements List, F30602-76-C-0233, CLIN 002, A001 R&D Status Reports.

6. JOCIT Compiler User's Manual, Computer Sciences Corporation, Revised August 1974.

7. C. Gannon, JAVS Acceptance Tests for RADC, General Research Corporation IM-1998, October 30, 1975.

8. D. M. Andrews, J. P. Benson, Software Quality Laboratory User's Guide, General Research Corporation CR-5-720, February 1977.

9. S. Saib et al., Advanced Software Quality Assurance Final Report, General Research Corporation CR-6-720, March 1977.